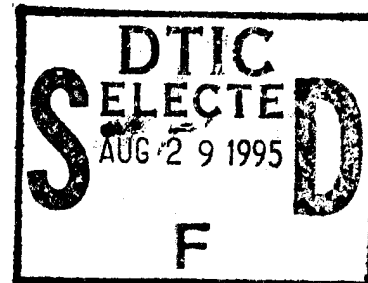# NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

FITTING DATA USING PIECEWISE G¹ CUBIC BÉZIER CURVES

by

Edward J. Lane

March, 1995

Thesis Co-Advisors: Richard Franke
                              Carlos F. Borges

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE March, 1995.	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE FITTING DATA USING PIECEWISE G^1 CUBIC BÉZIER CURVES	5. FUNDING NUMBERS
6. AUTHOR(S) Edward J. Lane	

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*
A method is described for least squares fitting an ordered set of data in the plane with a free-form curve with no specific function or parameterization given for the data. The method is shown to be effective and uses some techniques from the field of Computer Aided Geometric Design (CAGD). We construct a piecewise G^1 cubic Bézier curve from cubic curve segments which have as their initial end points, or knot points, some of the data points. The parameters for the curve are: the knot points, the angles of the tangent vectors at the knot points, and the distances from each knot point to the adjacent control points. The algorithm is developed and three solution curves are presented: Globally Optimized Only (GOO), Segmentally Optimized Only (SOO), and Segmentally then Globally Optimized (SGO).

14. SUBJECT TERMS Free-form Curve, Piecewise G^1 Cubic Bézier Curve, Knot Point, Control Point, Least Squares.	15. NUMBER OF PAGES 88
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited.

FITTING DATA USING PIECEWISE
G^1 CUBIC BÉZIER CURVES

Edward J. Lane
Lieutenant, United States Navy
B.S., Jacksonville University, 1987

Submitted in partial fulfillment
of the requirements for the degree of
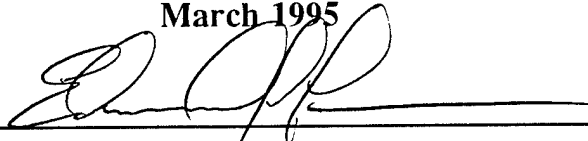
MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
March 1995

Author: _____
Edward J. Lane
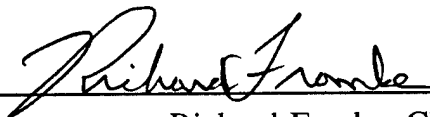
Approved by: _____
Richard Franke, Co-Advisor

_____
Carlos F. Borges, Co-Advisor

_____
Richard Franke, Chairman
Department of Mathematics

Accesion For		
NTIS CRA&I	☑	
DTIC TAB	☐	
Unannounced	☐	
Justification		
By		
Distribution /		
Availability Codes		
Dist	Avail and / or Special	
A-1		

iii

ABSTRACT

A method is described for least squares fitting an ordered set of data in the plane with a free-form curve with no specific function or parameterization given for the data. The method is shown to be eff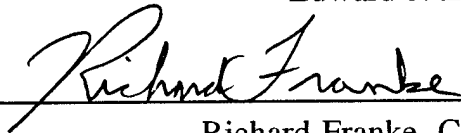ective and uses some techniques from the field of Computer Aided Geometric Design (CAGD). We construct a piecewise G^1 cubic Bézier curve from cubic curve segments which have as their initial end points, or knot points, some of the data points. The parameters for the curve are: the knot points, the angles of the tangent vectors at the knot points, and the distances from each knot point to the adjacent control points. The algorithm is developed and three solution curves are presented: Globally Optimized Only (GOO), Segmentally Optimized Only (SOO), and Segmentally then Globally Optimized (SGO).

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I wish to acknowledge my advisors, Dr. Richard Franke and Dr. Carlos F. Borges, for their counsel and patience throughout the research and writing of this thesis. Their knowledge of Mathematics provided great inspiration and motivation during what seemed at times an uphill struggle.

I also wish to acknowledge my Dad and the memory of my Mom. Their sacrifices while I grew up and the lessons in life they taught me entitles them to all the credit in my achievements.

I especially wish to acknowledge my wife Christine for the love she brings to my life. Her zest for living and pursuit of excellence are a driving force.

Finally, I wish to acknowledge the game of golf. Generally, an all around humbling game. Yet, at times, when club and ball meet in a pure fury, one of life's simplest pleasures.

I. INTRODUCTION

A. OBJECTIVES

This thesis is concerned with a practical method for fitting an ordered set of data in space with a free-form curve, with no specific function or parameterization given for the data. Problems such as this arise routinely in a variety of disciplines from the Arts to Engineering and Science. The techniques presented here are for data in the plane, $\Re^2$, but can be adapted to many dimensions.

The purpose of this study is to implement algorithms in MATLAB to further explore the feasibility of an automated routine which will examine an ordered set of data and, with possible user interaction, produce a fitted curve within specified conditions and tolerances. In considering the problem, we seek to fit a G^1 cubic Bézier curve to the ordered set of data using least squares approximation. Emphasis is given to those aspects of problem analysis and formulation leading to solution algorithms and procedures.

B. OVERVIEW

Polynomials are widely used for data approximation and curve fitting, primarily because they are relatively simple functions. Their sums, differences, and products are polynomials, as are their derivatives and integrals. Further more, a shift in the origin of the coordinate system or a scaling of the independent variable for a polynomial produces a polynomial (Carnahan, 1969).

According to Rivlin (1981), one of the most direct ways to approximate a function on an interval, or a finite set of points, is to obtain a polynomial which takes on the same values as the function at some points in the domain of the function. This is useful if we can show that a polynomial can provide a "good approximation" to a given function $f(x)$. By

1

"good approximation," we mean the ability to constrain the error of a polynomial approximation to the function to an arbitrarily small value. It turns out that justification exists in the form of the **Weierstrass approximation theorem** which we present here without proof (Ralston, 1965):

1.1 THEOREM. (Weierstrass approximation theorem)

If $f(x)$ is a continuous function on a finite interval $[a,b]$, then, given any $\varepsilon > 0$, there exists an $n [= n(\varepsilon)]$ and a polynomial $P_n(x)$ of degree n such that $\left| f(x) - P_n(x) \right| < \varepsilon$ for all x in $[a,b]$.

Given the assurance that some polynomial $p(x)$ does exist to approximate every continuous function $f(x)$, we now look to fit an ordered set of data points (x_i, y_i), which are assumed to satisfy $y_i = f(x_i)$ for some continuous function $f(x)$, by approximating $f(x)$ by a polynomial $p(x)$. One of the requirements we seek to enforce in fitting the data is that the process be unambiguous. Another is to find a fit which minimizes any deviations between the data points and the curve. Assuming the errors are negligible in one of the two measurements of our data, the usual criterion would be to minimize the sum of the squares of the error in the other, this is the linear least-squares principle and is commonly called a least squares fit.

Fitting a set of data by least squares has many benefits, one of which is the statistical principle of maximum-likelihood. The principle says, "If the measurement errors have a normal distribution and if the standard deviation is constant for the data, the fitted line by minimizing the sum of the squares is shown to have slope and intercept having maximum-likelihood of occurrence" (Mendenhall, 1990). Another benefit is that a unique solution for a given set of data is

2

guaranteed.

Now that we have established some fitting criteria for the polynomial, the next step is to decide upon the degree. For a set of data with $n+1$ data points, one strategy for choosing the degree of the polynomial is to fit some or all of the data points with a polynomial of degree at most n that interpolates the points. This is a poor strategy because, while it minimizes the distances between the curve and the data points, a higher degree polynomial amplifies errors in the input data. Another reason is that while the polynomial approximates the data to within some required degree of accuracy, higher degree polynomials have an inherent localized "bump" or oscillation effect (Gerald, 1989).

When the degree n of an interpolating polynomial $p(x)$ is large we encounter undesirable oscillations because there may be as many as $n-1$ maxima and minima. Further, as the number of points to be approximated gets larger and larger, the oscillations may also increase. In most cases, an intermediate degree, usually three, polynomial is the best choice.

A remedy to the undesirable effects of higher degree interpolating polynomials is to construct composite curves which fit lower degree polynomials to successive groups of data points. This process produces piecewise interpolating polynomial functions. Due to their flexibility, these functions are more widely used in least-squares fitting. However, although they can be continuous functions, they will usually have discontinuities in slope at the joining points of their successive segments. For most applications, this behavior is unacceptable and must be avoided.

To that end, consider a fitted piecewise interpolating polynomial $p(x)$ for function $y = f(x)$, and its points $x_{i-1} < x_i < x_{i+1}$. If we assume both y and the first

derivative $\dot{y}$ to have continuity in value at each point x_j along the polynomial, then the resulting piecewise function will have continuity of slope at all data points and be "smooth" everywhere.

Piecewise functions often involve segments of cubic polynomials. This is because cubic polynomials offer not only the opportunity to match up slopes but also curvature when joined. The most common of these functions are called cubic splines and are used extensively in approximation, interpolation, and data fitting. One disadvantage to cubic splines is that their interpolant derivatives may not agree with those of the function being approximated, even at the points joining the segments.

An alternative to the piecewise interpolating polynomial curve is to create a curve using approximation techniques that builds on its attractive qualities and does not, or at least is not required to, pass through all the points in the data set. Rather, some of the points are used to control the shape of the resulting curve. For such a curve, its x and y components are parameterized in terms of another variable t, for example, and equations for the points $(x(t),y(t))$ on the curve are called parametric equations. The variable t is called the parameter for the curve. One such curve that can be constructed in this manner and is of special interest is the Bézier curve.

C. PROBLEM STATEMENT

For a given ordered set of data in the plane,

$$S_i = (x_i,y_i) \ , \ i = 1,2,3,\ldots,n \ ,$$

we wish to find the curve that minimizes the sum of the squares of the deviations from each data point to the nearest point on the curve. In solving the problem, we will fit the data points S_i with a piecewise G^1 cubic Bézier curve by a

4

least squares criteria. We use MATLAB's "fmins" optimization routine to find three solutions to the problem: a globally optimized only (GOO) fit, a segmentally optimized only (SOO) fit, and a segmentally then globally optimized (SGO) fit.

D. RESEARCH METHODOLOGY

The research for this thesis was accomplished in five phases. First, information about the subject was gathered. Second, the computer algorithms of Holmes' (1993) were evaluated for operation and revised where applicable. Third, new algorithms were implemented. Fourth, performance of the algorithms using a variety of data sets presenting unique challenges was examined and results were tabulated. Lastly, conclusions were drawn and recommendations for further research were considered.

E. THESIS ORGANIZATION

This thesis is organized into five chapters. Chapter II is a discussion of the concepts and theory used to develop material introduced in Chapter III. The discussion covers some treatments found in Ross (1980), Farin (1990), and others. Chapter III is a discussion introducing Bernstein polynomials, Bézier curves, and other applicable topics. It follows treatments found in Gerald (1989), Farin (1990), and others. Chapter IV describes the implementation of algorithms to fit a G^1 cubic Bézier curve to an ordered set of data points in the plane. Chapter V contains results, conclusions and some recommendations for future work. An appendix is provided with some flow charts for the algorithm and the programs. Further, a tutorial is available if desired.

II. BACKGROUND

A. CONTINUITY OF FUNCTIONS

The study of calculus usually provides the first introduction to continuous functions. Recall that for a function f:

- the domain of f, written $dom(f)$, is the set S upon which f is defined.

- f is a rule or formula which assigns a unique value $f(x)$ to each $x \in dom(f)$.

We will be interested in functions f where the domain of f is a subset of the reals, $dom(f) \subseteq \Re$, and where $f(x) \in \Re$ for all $x \in dom(f)$.

In most cases, the domain of a function will be specified. However, when it is not, the domain is understood to be the natural domain or largest subset of $\Re$ on which the function is "real-valued" and well defined. As an example, $\{x \in \Re : x \neq 0\}$ is understood to be the natural domain of $f(x) = 1/x$ while we normally just write $f(x) = 1/x$. This leads us to the definition of a continuous function.

2.1 DEFINITION. Let f be a real-valued function whose domain is a subset of $\Re$. Then f is continuous at x_0 in $dom(f)$ if, for every sequence (x_n) in $dom(f)$ converging to x_0, we have $\lim f(x) = f(x_0)$. If f is continuous at each point of a set $S \subseteq dom(f)$, then f is said to be continuous on S. Function f is said to be continuous if it is continuous on $dom(f)$.

Definition **2.1** suggests that values $f(x)$ are close to $f(x_0)$ whenever the values x are close to x_0. We now introduce and prove a theorem about continuous functions which states this more formally.

7

2.2 THEOREM. Let f be a real-valued function whose domain is a subset of $\Re$. Then f is continuous at $x_0 \in dom(f)$ if and only if for each $\varepsilon > 0$ there exists $\delta > 0$ such that $x \in dom(f)$ and $|x-x_0| < \delta$ implies $|f(x)-f(x_0)| < \varepsilon$.

To prove **2.2**, suppose that its conclusion holds and consider a sequence (x_n) in $dom(f)$ such that $\lim x_n = x_0$. Now we must show that $\lim f(x_n) = f(x_0)$. So choose $\varepsilon > 0$. From **2.2**'s conclusion, there exists $\delta > 0$ such that when $x \in dom(f)$ and $|x-x_0| < \delta$ then $|f(x)-f(x_0)| < \varepsilon$. Since $\lim x_n = x_0$, there exists a number k such that $n > k$ implies $|x_n-x_0| < \delta$. It then follows that $n > k$ also implies $|f(x_n)-f(x_0)| < \varepsilon$. This proves $\lim f(x_n) = f(x_0)$.

For the second part, we assume that f is continuous at x_0 but that **2.2**'s conclusion fails to hold. This means there exists $\varepsilon > 0$ such that the implication "$x \in dom(f)$ and $|x-x_0| < \delta$ implies $|f(x)-f(x_0)| < \varepsilon$" fails for each $\delta > 0$. Particularly, when $\delta = 1/n$ the implication fails for every $n \in \mathbf{N}$. Therefore, for every $n \in \mathbf{N}$ there exists x_n in $dom(f)$ such that $|x_n-x_0| < 1/n$ and $|f(x_n)-f(x_0)| \geq \varepsilon$. Thus we have $\lim x_n = x_0$. But, since $|f(x_n)-f(x_0)| \geq \varepsilon$, we cannot have $\lim f(x_n) = f(x_0)$ for all n. However, this is contradictory to our assumption that f is continuous at x_0. Therefore, **2.2**'s conclusion must hold. ∎

Uniform continuity

We now introduce the definition of a uniformly continuous function:

2.3 DEFINITION. Let f be a real-valued function defined on

a set $S \subseteq \Re$. Then f is uniformly continuous on S if for every $\varepsilon > 0$ there exists $\delta > 0$ such that $x, y \in S$ and $|x-y| < \delta$ implies $|f(x) - f(y)| < \varepsilon$. When f is uniformly continuous on $dom(f)$, we call f uniformly continuous.

There are some important notions inferred by referring to a function as uniformly continuous. First, uniform continuity alludes to the function f and the set upon which it is defined. It makes very little sense to say that a function is uniformly continuous at a point. Second, looking at definition **2.3**, we note it is sometimes very useful to know when a $\delta > 0$ can be chosen to depend solely on $\varepsilon > 0$ and set S, rather than δ depending on the particular point x_0.

We now present and prove an important theorem on functions that are uniformly continuous:

2.4 THEOREM. If f is continuous on a closed interval $[a,b]$, then f is uniformly continuous on $[a,b]$.

To prove **2.4**, assume that f is not uniformly continuous on $[a,b]$. Then there exists $\varepsilon > 0$ such that for every $\delta > 0$ the implication "$|x-y| < \delta$ implies $|f(x) - f(y)| < \varepsilon$" fails. This means that for every $\delta > 0$ there exists $x, y \in [a,b]$ such that $|x-y| < \delta$, however, $|f(x) - f(y)| \geq \varepsilon$. This means for each $n \in \mathbf{N}$ there exists $x_n, y_n \in [a,b]$ such that $|x_n - y_n| < 1/n$, yet $|f(x_n) - f(y_n)| \geq \varepsilon$. From the study of bounded sequences, we know "every bounded sequence has a convergent subsequence", this is the **Bolzano-Weierstrass theorem**. This tells us in this case that a subsequence (x_{n_k}) of (x_n) converges. Further, if $\lim_{k \to \infty} x_{n_k} = x_0$, then $x_0 \in [a,b]$. Similarly, we would also have $\lim_{k \to \infty} y_{n_k} = x_0$. Now, since f is continuous at x_0, we have

$\lim_{k \to \infty} f(x_{n_k}) = \lim_{k \to \infty} f(y_{n_k}) = f(x_0)$. Thus we have $\lim_{k \to \infty} [f(x_{n_k}) - f(y_{n_k})] = 0$. But, since $|f(x_{n_k}) - f(y_{n_k})| \geq \varepsilon$ for all k, there is a contradiction. Hence, our assumption must be false, and we conclude that f is uniformly continuous on $[a,b]$. ■

The integrability of continuous functions on closed intervals is an important application of uniform continuity. For more information on this and other topics from analysis, see Ross (1980).

B. VECTOR SPACES

Most of us, at one time or another, have used the Cartesian coordinate system spaces, $\Re^2$ and $\Re^3$, to describe or investigate physical quantities such as position, velocity, and acceleration. These quantities are sometimes referred to as "geometrical vectors" or "directed line segments", so named because they "live" in a geometrical or physical space. It is assumed that the reader is familiar with these concepts and the operations of vector addition and scalar multiplication, and further, the concept of a vector space.

Let S be the set of scalars or real numbers. A vector space V will then be defined to be a set of elements $v_1, v_2, \ldots, v_n$, called vectors, such that for $s \in S$ and $v_1, v_2 \in V$, the operation of scalar multiplication produces a unique vector $sv \in V$, and the operation of vector addition produces a unique vector $(v_1 + v_2) \in V$. Further, for vectors $u, v, w \in V$ and scalars $r, s \in S$ the following properties are satisfied:

- the commutative and associative laws of addition.

- the distributive and associative laws of multiplication.

- the existence of an additive inverse.

- the existence of an additive identity and multiplicative identity.

For a vector space V so defined, we say V is closed under the operations of addition and scalar multiplication.

It is easy to show that $\mathfrak{R}^n$ is a vector space for any positive integer n. See Hill, (1990) and Ross (1980) for further information.

C. POINTS, VECTORS, AND CONVEX COMBINATIONS

When we write $\mathfrak{R}^n$, we mean Euclidean n-space. Euclidean n-space is the vector space as described above with the natural metric,

$$d(x,y) = \sqrt{(x_1-y_1)^2 + (x_2-y_2)^2 + \ldots + (x_n-y_n)^2} \quad ,$$

and inner product

$$x \cdot y = x_1 y_1 + x_2 y_2 + \ldots + x_n y_n \quad .$$

We will use certain conventions when working with points and curves in any vector space. For example, the space must have a coordinate system that does not affect any properties of the points or curves. In addition, the coordinate system must not influence any methods we may generate and employ.

While both points and vectors "live" in $\mathfrak{R}^n$, and may be described in similar notation such as n-tuples, there is an important distinction. For any two points p_1 and p_2 in a space, there is a unique vector v_{12} that is directed from p_1 to p_2. However, for vector v_{12}, there are many pairs of points p_i, p_j ; $i \neq j$ where $v_{12} = p_j - p_i$. To show this, consider two points p_1, p_2 which describe vector $v_{12} = p_2 - p_1$. If v_n is an arbitrary vector in the space, then $p_1 + v_n$, $p_2 + v_n$, the

11

translation of p_1, p_2, is another pair of points which also describe vector v_{12} since $v_{12} = (p_2 + v_n) - (p_1 + v_n)$. This is because vectors are invariant under translations while points are not.

Addition and subtraction of vectors is a well defined operation since vectors are invariant under translations. However, this is not true of points. Whereas subtraction is defined and produces a vector, addition is not defined since different coordinate systems would produce different "solutions" (Farin, 1990). Nonetheless, there are "addition-like" operations defined for points and these are called affine or barycentric combinations.

The term barycenter means center of gravity. A barycentric combination is a weighted sum of points such that the weights sum to one. For instance, point p,

$$p = \sum_{i=0}^{m} w_i \, p_i$$

where $p_i \in \Re^3$ and $\sum w_i = 1$, is a barycentric combination. Although p may appear to be the result of an undefined operation, pointwise addition, we can easily rewrite it as the sum of a point and vector,

$$p = p_0 + \sum_{i=1}^{m} w_i \, (p_i - p_0) \quad ,$$

which is defined.

There are certain barycentric combinations whose coefficients w_i not only sum to one, but are also nonnegative. These are called convex combinations. A convex combination will always lie inside the boundary of the polygon made by connecting the points which make up the convex combination. This is illustrated in Figure 1.

In addition, the set of points composed of all convex combinations of a point set is known as the convex hull of the point set. Such a set, the convex hull, is also a convex set

12

point p is a convex combination of
points p0, p1, p2, p3, and p4

the convex hull formed by
p5, p6, p7, and p8
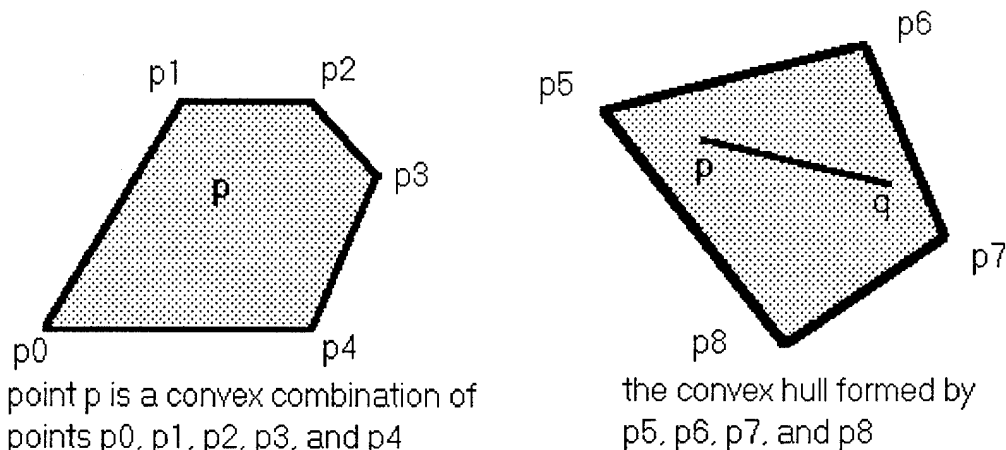
Figure 1. A convex combination and a convex hull.

and is distinguished by the property that all points on a straight line joining any two points in the set is completely contained within the set. This is also illustrated in Figure 1.

D. AFFINE MAPS

Consider a point $p \in \Re^3$ and a mapping μ that maps p as follows:

$$\mu p = Mp + v \qquad\qquad 2.1$$

where M is a 3×3 matrix and $v \in \Re^3$ a vector. A map as described in Equation **2.1** is called an affine map. Affine maps are the most common transformations used to position and scale objects in computer graphics and computer aided design (CAD) (Farin, 1990). The definition follows:

2.5 DEFINITION. An affine map is a map μ that maps $\Re^3$ pointwise into itself and leaves barycentric combinations invariant. It may be composed of rotations, scalings, shears, and translations. Additionally, an affine map leaves ratios of collinear points unchanged and preserves parallels.

We can show that barycentric combinations are preserved

13

under affine maps by writing p as $\sum w_i p_i$ and recalling $\sum w_i = 1$. The proof is as follows:

$$\mu\left(\sum w_i p_i\right) = M\left(\sum w_i p_i\right) + v$$

$$= \sum w_i M p_i + \sum w_i v$$

$$= \sum w_i (M p_i + v)$$

$$= \sum w_i \mu p_i \quad .$$

Thus, we see that if μ is an affine map and,

$$p = \sum w_i p_i \; ; \quad p, p_i \in \Re^3 \quad ,$$

then

$$\mu p = \sum w_i \mu p_i \; ; \quad \mu p, \mu p_i \in \Re^3 \quad .$$

We note that affine maps may be combined to form more complex maps or decomposed into a series of simpler maps.

E. LINEAR INTERPOLATION

The term interpolation refers to the constraint that an approximated curve or surface fitted to a set of points pass through the points. Consider a set of points p in $\Re^3$ defining a line such that:

$$p = p(t) = p_1 + t (p_2 - p_1) \; ; \quad t \in \Re \quad . \qquad 2.2$$

The line passes through p_1 when $t = 0$ and through p_2 when $t = 1$. For $0 \leq t \leq 1$, point p lies on the line between p_1 and p_2. For all other values of t, point p lies on the line outside of the interval between p_1 and p_2. Hence, we see that the equation for p, as written in Equation **2.2**, is a barycentric combination of two points.

Intuitively, we may write t as $t = 0 + t(1-0) \; ; \quad t \in \Re$,

14

also a barycentric combination. This shows that t relates to 0 and 1 in the same manner as p is related to p_1 and p_2, a barycentric combination, see Figure 2. Additionally, we have mapped three points from the real line, $0, t, 1$, to three points, p_1, p, p_2, in 3-space. By definition, this is an affine map. Further, we note, without proof, that in the process the ratios among the points, $0, t, 1$ and p_1, p, p_2, in their respective spaces has been preserved (Farin, 1990).
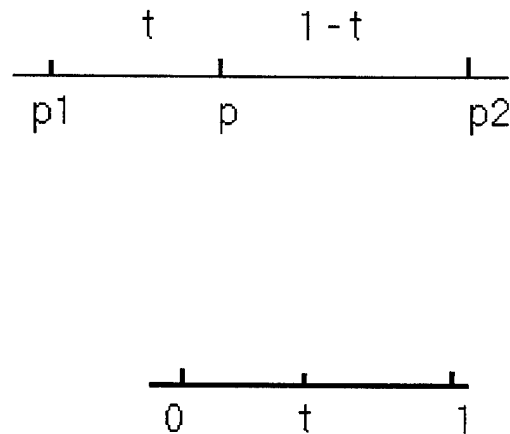
Figure 2. Linear interpolation.

We refer to linear interpolation as being affinely invariant. Affine invariance is the property in a curve or surface generation scheme that allows computation of a point on the curve or surface before or after an affine map is applied to the point.

While we mapped the interval [0,1] to $[p_1, p_2]$, we could just as well have chosen an arbitrary interval $[x, y]$. To see this, consider the interval $[x, y]$ as an affine map from [0,1]. Letting $t \in [0,1]$ and $s \in [x, y]$, our map is $t = (s-x)/(y-x)$. Then, from $p(t) = p_1 + t(p_2 - p_1)$, we now have

$$p(s) = \frac{y-s}{y-x}p_1 + \frac{s-x}{y-x}p_2 \ .$$

Thus, $0,t,1$ and x,s,y and p_1,p,p_2 are all in the same ratio. This shows that linear interpolation is invariant under affine domain transformations.

F. PIECEWISE LINEAR INTERPOLATION

Consider a polygon P composed of a series of line segments connecting points $p_0,p_1,p_2,\ldots,p_n \in R^3$. These line segments each interpolate between points p_i,p_{i+1} . Hence, P is referred to as the piecewise linear interpolant PL to the points p_i . When points p_i lie on a curve c,

$$P = PLc$$

is the piecewise linear interpolant to c .

For an affine map μ that maps curve c onto curve μc, the piecewise linear interpolant to μc is

$$PL\mu c = \mu PL c \ , \hspace{3cm} 2.3$$

which is the affine map of the piecewise linear interpolant. From **2.3**, we see that piecewise linear interpolation exhibits the property of affine invariance.

Also exhibited by piecewise linear interpolation is the variation diminishing property. This is the property that a piecewise linear interpolant to a curve has no more intersections with a plane than the curve. This is demonstrated in Figure 3. As can also be seen in the figure, the line joining points p_0 and p_1 can cross a plane running between them in at most one point. However, a curve connecting the two points can cross the same plane at many points, see (Farin, 1990).
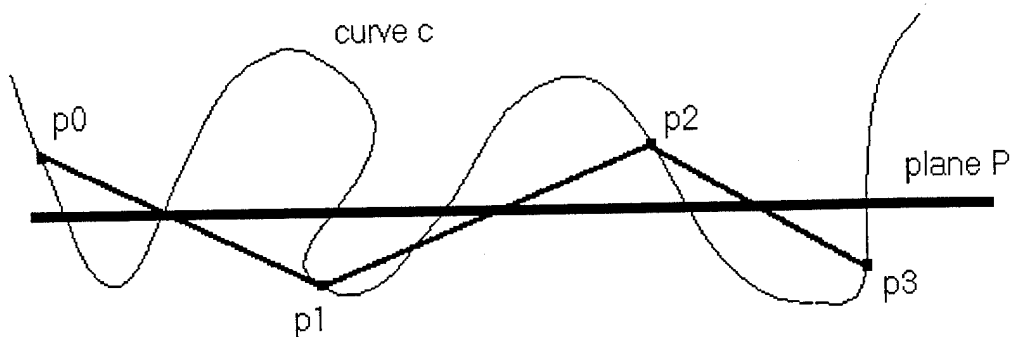
the piecewise linear interpolant to curve c has no more intersections with plane P than the curve does

Figure 3. The variation diminishing property.

G. FUNCTION SPACES

Recalling the discussion on vector spaces, we now note that some of the same properties may hold under more abstract conditions than were previously mentioned. We will therefore define a "vector space" to be a set of objects in which those properties mentioned hold and a "vector" will simply be one of the objects in the space. Consequently, a "vector" may hold little, if any, resemblance to a "directed line segment".

We first consider the set $C[a,b]$ of all real-valued continuous functions defined over the interval $[a,b]$. For the function $f(x) = 1/x$, f is in $C[1,2]$ because the function is defined on the whole interval $[1,2]$. However, f is not in the set $C[-1,1]$, because the function is undefined at $x = 0$. Letting f and g be elements of $C[a,b]$, and s be a real number or scalar, we define addition and scalar multiplication by $(f+g)t = f(t)+g(t)$ and $(sf)t = sf(t)$ for all $t \in [a,b]$. It is easy to see that $f+g$ and sf are in $C[a,b]$ and $C[a,b]$ is closed under addition and scalar multiplication. Further, it can be shown that $C[a,b]$ forms a "vector space" and its "vectors" are functions.

We end with an example of a space from $C[a,b]$ which will

17

be of interest in the next chapter.

Example. Consider P_n, the set of all polynomials of degree less than or equal to n. For polynomials $p, q \in P_n$, where $p = a_o + a_1 x + \ldots + a_n x^n$ and $q = q_0 + q_1 x + \ldots + q_n x^n$, let us define addition and scalar multiplication by:

$$p + q = (a_0 + b_0) + (a_1 + b_1) x + \ldots + (a_n + b_n) x^n$$

$$sp = (sa_0) + (sa_1) x + \ldots + (sa_n) x^n \quad .$$

From these definitions, we see that P_n is closed under addition and scalar multiplication. In addition, it is easy to show that P_n is a "vector space" (Hill, 1991).

III. BÉZIER CURVES

A. BERNSTEIN POLYNOMIALS

The expression

$$B_n(t) = \sum_{i=0}^{n} f\left(\frac{i}{n}\right)\binom{n}{i} t^i (1-t)^{n-i} \qquad 3.1$$

for $f(t)$ defined on the closed interval $[0,1]$ is the Bernstein polynomial of order n for the function $f(t)$. Polynomials of the form in Equation **3.1** are named after S. N. Bernstein who introduced them as part of an especially eloquent proof of Weierstrass' approximation theorem, see Davis (1963), Lorentz (1986), Rivlin (1981), or Ross (1980). The polynomials have many remarkable properties and have been linked to a variety of topics to include analysis, divergent series, moment problems, and probability. In referring to Bernstein's polynomials, Lorentz (1986) calls them "the most important and interesting concrete operators on a space of continuous functions". Our interest in them lies in their "good" approximation properties and their use as a basis for cubic Bézier curves.

We may rewrite Equation **3.1** as

$$B_n(t) = \sum_{i=0}^{n} f\left(\frac{i}{n}\right) B_i^n(t)$$

where the $B_i^n(t)$ are the Bernstein basis polynomials

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i} \quad ; \quad \begin{array}{l} i = 0,1,2,\ldots,n \\ n = 0,1,2,\ldots \end{array} \qquad 3.2$$

Equation **3.2** is recognizable from probability theory as the probability density function for the discrete binomial distribution.

19

Properties of Bernstein Polynomials

We now introduce some of the important properties of Bernstein polynomials:

1. The polynomials exhibit pairwise symmetry over the interval $[0,1]$, with respect to t and $1-t$, and are also non-negative. Pairwise symmetry is shown by noting:

$$B_i^n(t) = B_{n-i}^n(1-t) \ .$$

Non-negativity can be seen by the terms in the expression for the $B_i^n(t)$.

2. For any valid t, $\sum_{i=0}^n B_i^n(t)$ is always one. This is shown as follows:

$$\sum_{i=0}^n B_i^n(t) = \sum_{i=0}^n \binom{n}{i} t^i (1-t)^{n-i} = (t+(1-t))^n = 1$$

Hence the polynomials form a partition of unity.

3. The polynomials satisfy a three-term recurrence relation:

$$B_i^n(t) = (1-t) B_i^{n-1} + (t) B_{i-1}^{n-1}$$

with $B_0^0(t) = 1$ and $B_i^n(t) = 0$; $i \neq 0,\ldots,n$. This is proven as follows:

$$
\begin{aligned}
B_i^n(t) &= \binom{n}{i} t^i (1-t)^{n-i} \\
&= \binom{n-1}{i} t^i (1-t)^{n-i} + \binom{n-1}{i-1} t^i (1-t)^{n-i} \\
&= (1-t) B_i^{n-1}(t) + (t) B_{i-1}^{n-1}(t) \ .
\end{aligned}
$$

4. As with all polynomials, their sums, differences, and

products are polynomials, as are their derivatives and integrals. And, if the coordinate system origin is shifted or the independent variable scaled, the transformed polynomials $B_n(t+a)$ and $B_n(st)$ are also polynomials.

B. BÉZIER POLYNOMIAL CURVES

Bézier curves and surfaces are attributed to two men who developed them independently while working for rival French automobile companies. P. de Casteljau worked for Citroën around 1959 while P. Bézier worked for Rénault around 1962. Both applied the Bernstein polynomials to computer aided design (CAD) systems used for designing the unique curves and shapes required for automobile body panels. De Casteljau's work was held as proprietary whereas Bézier's design software system, called UNISURF, was published. Thus the curves and surfaces bear Bézier's name. In 1975, W. Boehm obtained two technical reports attributed to de Casteljau and his work has since gained prominence (Farin, 1990). The de Casteljau algorithm for generating a degree n Bézier curve b^n is as follows:

de Casteljau algorithm
Given: $p_0, p_1, \ldots, p_n \in \Re^3$, $t \in \Re$;
 for $m = 1, 2, 3, \ldots, n$, and $i = 0, 1, 2, \ldots, n-m$,
set

$$b_i^m(t) = (1-t)b_i^{m-1}(t) + (t)b_{i+1}^{m-1}(t) \quad ; \text{ where } b_i^0(t) = b_i = p_i .$$

At parameter value t, the point $b_0^n(t)$ is on the curve b^n.

Figure 4 displays the results of the algorithm. Connecting the points $b_0, b_1, b_2, \ldots, b_n$ by straight lines forms a polygon known as the control or Bézier polygon for the curve
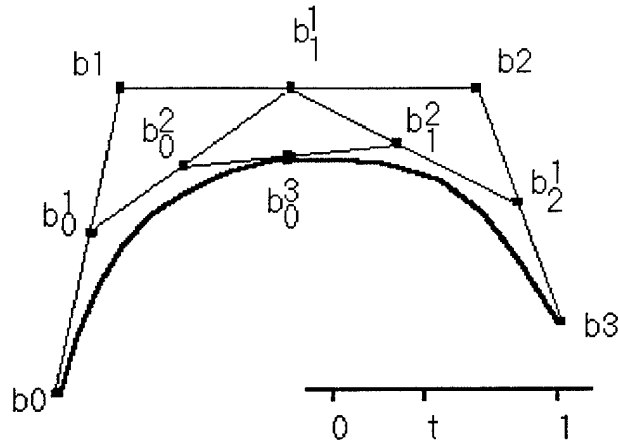
Figure 4. The de Casteljau algorithm.

b^n. The b_i, the vertices of the polygon, are called control or Bézier points. The figure shows a Bézier curve of degree three or the cubic case. We note the curve is tangent to the first and last polygon segments and that it is contained by the control polygon. This will always be true. Lastly, we see that the point $b_0^n(t)$ is on the curve at parameter t as expected.

We remark that the appearance of Figure 4 also suggests the points $b_i^m(t)$, may be found using a tabular scheme having triangular form. This is referred to as the **de Casteljau scheme** and will be investigated further when we discuss subdivision.

For a Bézier curve b^n, with $n+1$ control points $b_n = (x_i, y_i)$; $i = 0, \ldots n$, we can define the curve parametrically by setting

$$x(t) = \sum_{i=0}^{n} x_i B_i^n(t) \qquad y(t) = \sum_{i=0}^{n} y_i B_i^n(t) \quad . \qquad 3.3$$

for $0 \leq t \leq 1$, where the $\sum_{i=0}^{n} B_i^n(t)$ are the Bernstein basis polynomials. (The Bernstein polynomials serve as a blending or

22

basis function for the curve.) Expanding the expressions in **3.3**, we see a Bézier polynomial curve has the following parametric form:

$$x(t) = (1-t)^n x_0 + n(1-t)^{n-1}(t) x_1 + \ldots + n(1-t)(t)^{n-1} x_{n-1} + (t)^n x_n ,$$

$$y(t) = (1-t)^n y_0 + n(1-t)^{n-1}(t) y_1 + \ldots + n(1-t)(t)^{n-1} y_{n-1} + (t)^n y_n .$$

We note $(x(0), y(0)) = b_0$ and $(x(1), y(1)) = b_3$, again proving that the curve passes through the endpoints of the control polygon formed by the b_n as was stated earlier.

Characteristics of a Bézier Curve

We now present some of the notable characteristics of a Bézier curve:

1. Invariance under affine transformations of the points. This is inherited from the de Casteljau algorithm which is a series of iterated linear interpolations or, more to the point, affine maps. The functional feature of this is as follows- whether we compute the points $b^n(t_j)$ and then apply an affine map to them individually, or simply apply the affine map to the control polygon and evaluate the polygon at the values t_j, the result is the same.

2. Invariance under affine transformations of the parameters. Recall the transition between the arbitrary interval $[x, y]$ and the interval $[0, 1]$ is an affine map and was done by introducing a parameter s, $x \leq s \leq y$, and letting $t = (s-x)/(y-x)$, where $0 \leq t \leq 1$. Since the de Casteljau algorithm uses ratios only, the interval is irrelevant. Thus a Bézier curve may be defined on an arbitrary interval.
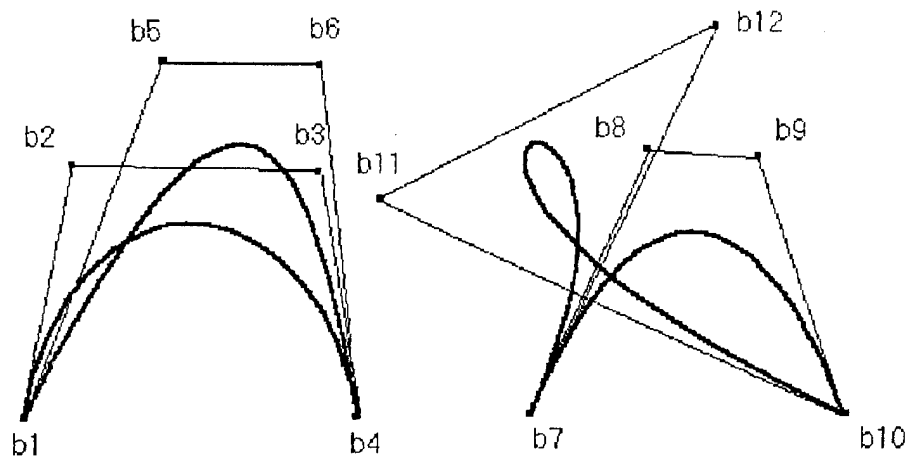
The control polygon determines the shape of the curve. Here we see the different curves formed from polygons beginning at points b1 and b4, and b7 and b10.

Figure 5. Various curves influenced by control point location.

3. Pseudo-local control. A change in control point locations has a fairly predictable effect on the curve. This is because control points have the most influence on the curve at the point $t = i/n$ where the Bernstein polynomial attains its maximum value. See Figure 5.

4. Only the first and last points or vertices of the control polygon are on the curve, see Figure 5. This is referred to as endpoint interpolation. In the case of a composite curve, the end points of the segments are interpolated on the curve.

5. They satisfy a convex hull property. At no time in the de Casteljau algorithm do we construct points outside the convex hull of the b_i because every intermediate point b_i^n is a convex combination of points. As a consequence of this property, a Bézier curve never oscillates wildly away from its defining control points.

24

6. Linear precision. This is another consequence of the convex hull property. Suppose the polygon's vertices b_i are distributed along a straight line joining points p_1 and p_2. Using the identity

$$\sum_{i=0}^{n} \left(\frac{i}{n}\right) B_i^n(t) = t \ ,$$

for points b_i, we find that

$$b_i = (1 - \frac{i}{n}) p_1 + \frac{i}{n} p_2 \quad ; \quad i = 0, \ldots, n.$$

The curve formed by this polygon will reproduce the straight line between p_1 and p_2.

7. The derivative of a Bézier curve is another Bézier curve. This is proven by starting with the derivative of a Bernstein polynomial:

$$\frac{d}{dt} B_i^n(t) = n(B_{i-1}^{n-1}(t) - B_i^{n-1}(t)) \quad .$$

We can then determine the derivative of a curve b^n as follows:

$$\frac{d}{dt} b^n(t) = n \sum_{k=0}^{n} (B_{k-1}^{n-1}(t) - B_k^{n-1}(t)) b_k$$

where b_k is a Bézier point. Since $B_k^n(t) = 0 \quad ; \quad k \notin \{0, \ldots, n\}$, we have

$$\frac{d}{dt} b^n(t) = n \sum_{k=1}^{n} B_{k-1}^{n-1}(t) b_k - n \sum_{k=0}^{n-1} B_k^{n-1}(t) b_k \ .$$

Reindexing and factoring we get

$$\frac{d}{dt}b^n(t) = n\sum_{k=0}^{n-1}(b_{k+1}-b_k)B_k^{n-1}(t) \ ,$$

which is the derivative of the curve b^n.

8.   The curve is tangent to the first and last segments of the control polygon. The m'th derivative of the first and last points of a Bézier curve are given by

$$\frac{d^m}{dt^m}b^n(0) = \frac{n!}{(n-m)!}\sum_{i=0}^{m}(-1)^{m-1}\binom{m}{i}b_i \ ,$$

$$\frac{d^m}{dt^m}b^n(1) = \frac{n!}{(n-m)!}\sum_{i=0}^{m}(-1)^{i}\binom{m}{i}b_{n-i} \ .$$

We see therefore that for a curve b^n, the first derivatives at the endpoints, $\dot{b}(0) = n(b_1-b_0)$, $\dot{b}(1) = n(b_n - b_{n-1})$, depend upon the first and last segments of its control polygon. Similarly, we could show that the second derivative at the end points is determined by the first and last two segments and, in general, the m'th derivative at an endpoint is determined by its m adjacent control points (Farin, 1990).

C.   CONTINUITY CONDITIONS

Thus far, most of our discussion of Bézier curves has centered on a single Bézier curve segment. However, for many applications, the need arises to piece or blend together segments of several curves to form a composite curve. In these cases, maintaining some type of continuity between the joined curve segments is usually desirable. Parametric continuity of order m, denoted C^m, results when the component functions of a parametric curve are m times differentiable with respect to the parameter and its given interval $[a,b]$. A curve has geometric continuity of order m, G^m, when it is m times differentiable with respect to arc length.

26
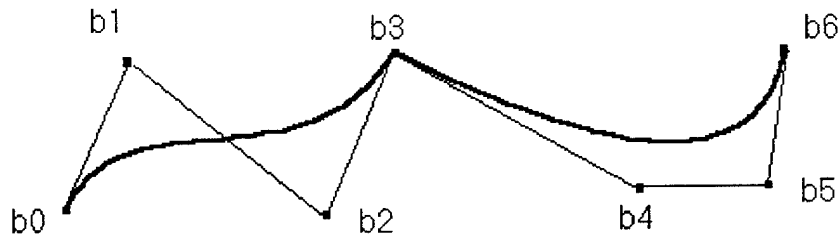
(a) parametric continuity order zero, C^0.
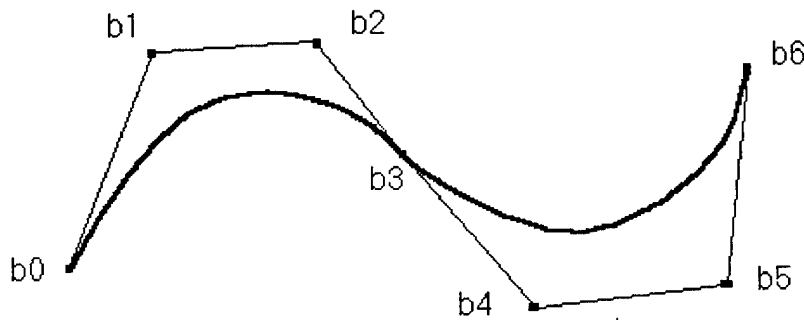
(b) parametric continuity order one, C^1.

Figure 6. Continuity conditions at joining points of curves.

For a composite curve to achieve C^0 continuity, it is sufficient to require one end control point from each of the successive segments to be a common point. For C^1 continuity, the end slope of one segment will be required to equal the starting slope of the succeeding segment. This means that for the successive segments of the composite curve the joining point between the segments is collinear with its adjacent control points. Figure 6 demonstrates these situations. However, although it does guarantee a continuously varying tangent, the collinearity of three control points is insufficient to guarantee C^1 continuity. This is because C^1 continuity relies on an interplay between range and domain. Hence, without the curve's domain information, statements on differentiability cannot be made. The absolute value function for a parametric curve $x = t^3$, $y = |x|$, $t \in [-1,1]$, is a good

27

example. This curve is C^1 but lacks continuity of the tangent at the origin.

For most applications, a curve which has less restrictive continuity conditions, such as G^1 continuity, continuously varying tangent with respect to arc length, is adequate. Moreover, there is rarely a need to require better than G^2 continuity, or continuously varying curvature. This, in turn, means little requirement for curves of higher order than cubic. (Pratt, 1986)

D. SUBDIVISION OF A BÉZIER CURVE

Subdividing or splitting a curve is characterized by replacing one curve with two or more curve segments of the same type such that the graph of the resulting composite curve is identical to that of the original. This is just a reparameterization or parameter transformation of the curve. Thus, for a Bézier curve b^n defined on the interval $[0,1]$, we now look to find two curves defined on the intervals $[0,k]$ and $[k,1]$.

We begin by looking at the interval $[0,k]$. If we define a local parameter $q = t/k$ on the interval, we see that $q = 0$ corresponds to $t = 0$, and $q = 1$, to $t = k$. Consequently, we have unknown points $k_0, k_1, \ldots, k_n$, corresponding to a Bézier polygon on the interval $[0,k]$ which defines a Bézier curve k^n. Further, the curve is clearly a part of the original curve b^n. To find the points k_i of the new polygon for the curve k^n, we must look at the relationship between the unknown k_i and the known b_i.

Since k^n and b^n are from the same polynomial curve, their derivatives evaluated at $q = t = 0$ must coincide. We now recall that the endpoint derivative of a Bézier curve is dependent only on the nearby control points. So to find the

m^{th} derivative of a curve b^n, we need points $b_0, \ldots, b_m$. Further, if we look at the first $m+1$ control points of the curves k^n and b^n as control polygons of two degree m Bézier curves, we find that the curves are identical.

Because the curves agree in all derivatives up to order m at $q = t = 0$,

$$k_0^m(q) = b_0^m(t) \quad ; \quad \textit{for all } q, t .$$

This expression also holds when $q = 1$ or $t = k$, that is,

$$k_0^m(1) = b_0^m(k) .$$

Since the endpoints of the control polygon for a Bézier curve are interpolated, we have $k_0^m(1) = k_m = b_0^m(k)$ and have established the unknown k_i.

Another approach to finding the unknown k_i uses the tabular scheme, the **de Casteljau scheme**, mentioned earlier in section B of this chapter. The form is as follows,

$$
\begin{array}{llll}
b_0 & & & \\
b_1 & b_0^1 & & \\
b_2 & b_1^1 & b_0^2 & \\
b_3 & b_2^1 & b_1^2 & b_0^3 \quad ,
\end{array}
$$

where the points b_i ; $i = 0, 1, \ldots, n,$, are the control points of the curve b^n. To find the unknown k_i, we simply pick off the elements of the main diagonal, the b_0^m ; $m = 0, 1, \ldots, n$. This is because the k_i, are just linear interpolants of the points b_i.

Graphically, we start by placing the points b_i in the first column of the table. Then, the subsequent entries in the table are found by blending the entry directly to the left

with the one to the left and above. This is comparable to the method of Chaikin (Cavaretta, 1989) and the iterated interpolation algorithms of Aitken and Neville. However, where Neville uses the same blending scheme, Aitken builds the table by blending the entry to the left and the first entry from the column to the left (Burden, 1981).

IV. IMPLEMENTATION

A. INITIAL GUESS ROUTINE

The first stage of the fitting algorithm is the assembly of an Initial Guess (IG) curve. To begin, a set of ordered data is placed into a 2 x n matrix or array, usually in a MATLAB file (ie. data.m). The user then reads in the data set, call it **Q**, and program "iguess" is called with **Q** as the argument. The user is prompted for the number of knot points or knots, **P**, which will initially be a subset of **Q**. Additionally, the user is prompted for the knot positions, **k**. These can either be manually entered or, by default, chosen by "iguess".

The ultimate goal of knot selection is to obtain a good fit for **Q** (Foley, 1989) with a minimum number of knots. This will be discussed further in Chapter 5. The subroutine "knots", called by "iguess" with arguments **Q** and **k**, picks out the initial set of knot points, **P**.

The **P** are arguments to subroutine "dist" which returns a set of distances, **dt**. The **dt** are computed using a standard formula for the distance between two points, successive knot points in this case, and are multiplied by one third. These distances will be used to obtain the initial locations of the interior control points in the IG curve's control polygon.

Next, the angle(s), **ang**, for the unit tangent vector at each knot point is/are calculated and returned when "iguess" calls subroutine "tang". The unit tangent is actually estimated by subroutine "unitv" which fits a parametric quadratic curve, using chord length parameterization, to five data points as follows: if the knot point is the first or last data point of **Q**, thus an end point for the curve, the five points will be the first or last five points, respectively, from **Q**; if the knot point is an interior data point of **Q**, then the five points will be the knot point and the two data points

on each side of it. The angles of the unit tangents are used
to indicate the direction from the knot point to the adjacent
interior control points.

Next, subroutine "ctpts" is called with arguments **P**, **ang**,
and **dt**. The subroutine reduces the angles into their x and y
components and multiplies them by the proper **dt**'s. These
quantities are added to and subtracted from the appropriate
knot points to find the set of control points, **C**, for the
curve which are then returned to "iguess".

Finally, "iguess" sends **Q**, **C**, and **P** as arguments to
"pltC". This subroutine plots the cubic Bézier IG curve along
with its control polygon and the points in **Q** for analysis.
Also, the parameters for the IG curve (**P**, **ang**, and **dt**) are
assembled into a composite vector **xi**, called IGC (for initial
guess curve) in "iguess", and returned along with **k** to the
user.

B. SEGMENT-WISE OPTIMIZATION ROUTINE

The   next   stage   of   the   algorithm   is   segment-wise
optimization of the IG curve producing what will be referred
to as a Segmentally Optimized Only (SOO) curve.   This is
accomplished by optimizing the set of control polygons for the
distances, **dt**, which best position the interior control points
to define a curve that produces minimum distance error between
itself and the data points for the segment.   The reason we
choose the **dt**'s is two-fold.   First, if the knots, **P**, were
selected properly, they will be positioned to produce a curve
which mimics the progression of the ordered data. Second, the
tangent vector to the curve at each knot will not change at
this   stage.   Further,   recall   the   control   points, **C**,   are
determined from **P**, **ang**, and **dt**, and, it is the control points
which govern the shape and behavior of the curve locally.

The user initiates segment optimization for the best **dt**'s
by calling the routine "segop". With arguments of **k**, **Q**, and

32

vector **xi** (IGC from "iguess"); "segop" first separates **xi** into its subcomponents via "ktangdt". Next, "segop" calls subroutine "bstdst" which uses the MATLAB optimization routine "fmins" to optimize the segments.

1. Optimization Routine

The purpose of MATLAB's optimization routine "fmins" is to minimize a function of several variables. The algorithm is based on the Nelder-Mead simplex search method (Nelder, 1965). In their paper, Nelder and Mead noted the applicability of an idea by Spendley et al. (1962) to the problem of minimizing a mathematical function of several variables. The notion was to track the operating conditions of a system by evaluating its output at a set of points, thus forming a simplex in the space of operations. By continuously reflecting one point in the hyperplane of the remaining points and forming new simplices, optimality could be achieved. The method is not based upon gradients nor quadratic (second-order derivative) forms. Rather, it is a highly opportunistic direct search method relying only on the assumptions of continuity and a unique minimum in the area of search. At no stage of the algorithm is a record of past positions kept. For more information, see (Nelder, 1965).

For the call x = fmins('func',x0), MATLAB returns a vector x which locally minimizes func(x) near x0. The term 'func' represents a string containing the name of the function to be minimized. For x = fmins('func',x0,options) and x = fmins('func',x0,options,[],p1,p2,...), the routine again returns local minimizer x. However, now the routine uses a vector of control parameters, 'options', for the algorithm. Some of the 'options' may be termination criteria for x, termination criteria for func(x), a maximum number for iterations of the algorithm, and so on. The routine may use 'options' and possibly some of up to ten potential arguments

to be passed on to the objective function, func(x,p1,p2,...). The argument in the fourth position of the third expression, the dummy argument, [], provides compatibility with routine "fminu" found in MATLAB's optimization toolbox. (Math Works, 1992)

When subroutine "bstdst" calls "fmins" to optimize the **dt**'s for each cubic segment in the composite curve, "fmins" is sent the string 'opdist', standing for subroutine "opdist", which will be the objective function. Also sent to "fmins" are: the **dt**'s for each segment (one segment at a time), some control parameters for "fmins", the dummy argument mentioned earlier, and three arguments pertaining to the applicable segment being optimized to pass to "opdist". The three arguments are: the subset of data points from **Q**, the two knot points, and the two angles for the tangents at the knots.

Subroutine "opdist" passes the three arguments to "ctpts" which returns the control points of the segment. Together, all the control points for the segment are sent as one argument to the program "NearestPoint" which receives as its other argument the points from the subset pertaining to the applicable segment from **Q**, sent one at a time. "NearestPoint" is an program written by Schneider (1990), modified by Dr. Carlos F. Borges to enable MATLAB to interface **C** routines, obtained from "Solving the Nearest Point-on-Curve Problem" and "A Bézier Curve-based Root-Finder".

2. Finding the Distance from a Data Point to a Curve

"NearestPoint" solves the following problem in the plane: for a given parametric curve $C(t)$ and a point p, find the closest point on the curve C to point p. Restated, the task is to find the value of parameter t where the distance between p and $C(t)$ is minimized. We begin by noting that a line segment joining p to $C(t)$, the length of which we seek

to minimize, will be perpendicular to the tangent of the curve at $C(t)$. Therefore, we will seek a solution to the equation

$$[C(t) - p] \cdot \dot{C}(t) = 0 . \qquad 4.1$$

In our case, $C(t)$ is a cubic Bézier curve,

$$C(t) = \sum_{i=0}^{n} k_i B_i^n(t) , \quad t \in [0,1]$$

where the k_i's are the control points and the $B_i^n(t)$'s are Bernstein polynomials. Expressing the derivative of $C(t)$ in Bézier form, we find the tangent for the curve to be

$$\dot{C}(t) = n \sum_{i=0}^{n-1} (k_{i+1} - k_i) B_i^{n-1}(t) .$$

Since $C(t)$, is degree three, we have $C(t) - p$ also degree three, and $\dot{C}(t)$ which is degree two. Therefore, Equation **4.1** is of degree five, generally. This means the problem boils down to one for which there is no closed form for a solution: finding the roots of a fifth degree polynomial. Thus, we turn to Schneider's technique and solve for the roots by using a recursive algorithm after first converting the equation to Bézier form. Once found, the roots are then evaluated to find the points on the curve $C(t)$ and the distances between these points and the point p is subsequently calculated. Further, the distances between the end points of the curve and the point p are calculated and then all of the distances are compared for a minimum. Thus, the parameter value t and the point on the curve $C(t)$ closest to the point p are found, as desired. (Schneider, 1990)

"NearestPoint" returns the point on the Bézier curve closest to the individual data points from the subset of **Q** for the segment being optimized. The distance between these "near

points" and their corresponding data points, error, is calculated and summed over the entire segment. This error sum is returned to "fmins" by the objective function "opdist".

Once the error sum is minimized, the best **dt**'s (called bdt's in the subroutine), for the control points for each segment are returned to "segop". As the last step, "segop" assembles the composite vector **xi** (called SOC, standing for Segmentally Optimum Curve, in "segop") of parameters (**P, ang**, and the new **dt**), and returns it for the SOO curve.

The user calls routine "poplt" with arguments **xi** and **Q**. This routine plots the SOO curve, its control polygon, and the data points for analysis.

C. GLOBAL OPTIMIZATION ROUTINE

The last stage of the algorithm is to globally optimize the SOO curve producing what we call a Segmentally then Globally Optimized (SGO) curve. It begins when the user calls routine "globop" with arguments **xi, Q, t**, and **k**. The argument **t** is a toggle to let the routine know if the knot sequence **k** has been altered by inserting or deleting any knots. The string 'objf2', for subroutine "objf2", is sent to "fmins" via "globop" as the objective function for minimization. In addition, "fmins" is sent: the vector **xi** to be optimized, a vector of control parameters for the routine, the dummy argument, and **Q, t**, and **k** as a fixed parameters to be sent to "objf2".

The optimization process of "objf2" begins with the subroutine separating **xi** into its components (**P, ang**, and **dt**). These components are sent to "ctpts" which returns the control points, **C**, for the curve. Next, "objf2" calls subroutine "newk" with **Q** and **P** as arguments.

Since **Q** is ordered, the closest points on the curve must be ordered in a like manner. This results in the requirement to associate each data point from **Q** with a particular cubic

segment. Thus, we avoid the error of computing distances for a point to a closer incorrect cubic segment, as can happen when the data turns rapidly back upon itself, forms spirals, or makes a loop. Subroutine "newk" determines which data points will partition **Q** into the subsets associated with the various segments of the curve.

Initially, the points which divide the data set are the knot points **P** chosen in "iguess". (Their positions, **k**, are passed to "newk" via the global variable **dpkpc**.) In order to find the new dividing points, **Pn**, which will divide the data points, "newk" searches among the data points for the point having the minimum distance from a knot point as follows: for an interior knot, the search is among the data points on each side of the knot excluding the previous and subsequent knots; for the first and last knots, there is no search since the first and last should be the first and last. Throughout the optimization process, as the knots move, "newk" updates the knot sequence for each iteration passing the subscripts of the knots via **dpkpc**.

With the continuously updated knot sequence available, "objf2" calls subroutine "sod" to compute the sum of the squares of the distances between the points of **Q** and their nearest respective points on the segments of the curve. The square of the distances from the first and last data points to the first and last knot points, respectively, is computed directly to ensure that the curve starts near the first knot point and ends near the last knot point.

Subroutine "sod" receives arguments **C**, **Q**, and the updated knot sequence, **dpkpc** from "objf2". Using "NearestPoint", "sod" computes the distances between the "near points" on the curve and their corresponding data points and sums the squares of these distances. It then returns this sum to "objf2" to be added to the squares of the distances for the first and last points.

The composite vector **xi** (called GOC for Globally Optimized Curve in "globop") of parameters (**P, ang,** and **dt**) which minimize the total sum of the squares of the distances found in "objf2" for the entire curve is returned from "fmins" to "globop" which in turn, returns **xi** to the user. The user can then call "poplt", with **xi** and **Q,** as was done earlier, for analysis.

Note: An IG curve could be, and is sometimes, globally optimized in a likewise manner. We call this a Globally Optimized Only (GOO) curve. This is done for reference purposes usually.

D. SUPPLEMENTAL ROUTINES

There are generally two types of error encountered in the fitting process. The two types are: excessive distance error between the data points and the curve, and appearance error where the curve has developed an undesirable feature such as a cusp or corner where not desired. Hence, it may be determined that some alterations and corrections are necessary in the curve. In order to address these situations, the user is supplied with the following routines: "err", "insrtkt", and "rmvkt".

1. Distance Error Checking

Routine "err" enables the user to check distance errors between a curve and a set of data points. The tolerance or threshold of error is determined by the user. The arguments for "err" are: the composite vector **xi** of parameters (**P, ang, dt**) for the curve, **Q,** and knot positions, **k.**

To determine distance error, "err" first calls "ktangdt" to separate the composite vector into its subcomponents. Next, "err" sends the **P, ang,** and **dt** as arguments to "ctpts" which returns the control points, **C,** for curve. The **C,** along

with **Q** and **k**, are sent to "sod" which returns the sum of the errors. In turn, "err" returns the total distance error for the curve.

Routine "err" can determine error for an individual segment of the cubic Bézier curve as well. To do so, the user simply uses some of the subroutines previously described in this chapter and calls "err" with the applicable components.

2. Knot Insertion and Removal

Routine "insrtkt" enables a user to insert a new knot in the knot sequence of a Bézier curve without altering the shape of the curve. However, before initiating the routine, two questions need to be answered: (1) Upon which segment will the knot be inserted? and (2) At what point along the segment will the knot be inserted? The user calls "insrtkt" with a segment number, and distance along the segment (i.e. "1/2" for half the distance, "3/4" for three quarters of the distance, and so on...), composite vector **xi**, **Q**, and **k**.

The routine begins by separating the parameters of **xi** via subroutine "ktangdt". Next, "insrtkt" calls "ctpts" with the knot points, angles, and distances for the affected segment and "ctpts" returns the segment's control points. Next, "insrtkt" calls subroutine "fndpts" with the control points of the segment and the distance along the segment where the new knot point will be inserted.

Incorporating an interpolatory subdivision algorithm, "fndpts" computes the new set of control points for the segment. It then assembles the values of the new control points and returns them to "insrtkt". Note: The new control polygon will reproduce the cubic Bézier curve segment of the original polygon whose control points were just subdivided.

Next, "insrtkt" separates the new control points into their x and y components. Then, intercomponent distances are found, angles for the tangent at the new knot point computed,

39

and the distances from the new knot point to its adjacent control points computed. Finally, the knot points, angles, and distances are assembled and returned as a composite vector **xi**, along with the new knot positions **nk**. The user calls routine "poplt" with **xi**, and **Q** for a plot of the curve, the new control polygon, and the data points for analysis.

Removing a knot point from a curve is a less complicated process. It should be pointed out that in a set of n knot points, only knots two through n-1 should be removed. (The reason for this is obvious.)

The user calls routine "rmvkt" with inputs of which knot point is to be removed (i.e. "2" for the second, "3" for the third, and so on) and the composite vector **xi**. The routine separates the components of **xi** via "ktangdt". Next, "rmvkt" simply removes the knot point and its associated angles and distances from their respective "vectors" by dropping the appropriately indexed subcomponents.

To merge the components from the two affected segments into one segment, "rmvkt" calls "ctpts" with the knot points that were on each side of the removed knot point, the associated angles for the tangents, and the distances at those knot points. Next, "ctpts" returns the control points for the new "merged" segment.

Routine "rmvkt" separates the x and y components of the new control points and computes their intercomponent distances. These intercomponent distances are then used to compute the distances for the control points of the new segment. The computation is based on the ratios that would have occurred in the de Casteljau algorithm if the two segments being merged had came from one segment, see Figure 7. The control point distances are then inserted into the vector of distances and "rmvkt" combines the knot points, angles, and distances into a composite vector **xi** which is returned along with the new knot positions **nk**.

40
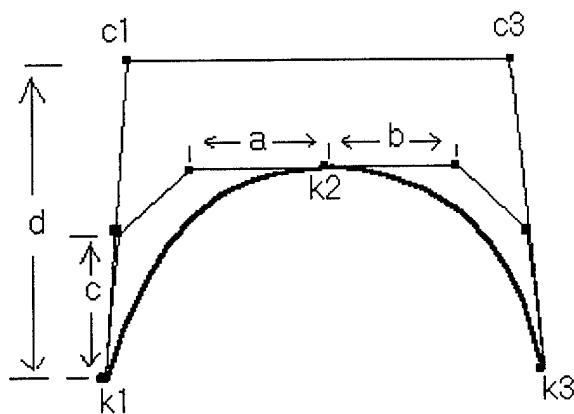
To find distance for new control point c1 from k1:

$$d = ((a+b)/a) \cdot (c) .$$

* can use similar technique to find distance for c3 from k3.

After removing k2, we treat the segment between k1 and k3 as if it had been one segment and been subdivided by de Castlejau's algorthim.

Figure 7. Finding new control point distances after removing a knot point.

The routine "poplt" may then be called to plot the curve, its new control polygon, and **Q**. Unlike the result obtained by inserting a knot, knot removal will likely change the curve slightly. This is due to the fact that in most cases the graph of two adjacent segments of a curve is not the graph of a single cubic curve.

V. RESULTS, CONCLUSIONS, AND RECOMMENDATIONS

A. RESULTS

Three data sets were selected with various fitting challenges. In the results that follow, we see the cubic Bézier curves fitted to those data sets and the errors of the fits. The curves for each data set are in the following order; initial guess (IG), globally optimized only (GOO), segmentally optimized only (SOO), and then segmentally and globally optimized (SGO). Rms error, in the form of arbitrary "units", representing distance summed between the curve and data points is noted for the various curves. For demonstration purposes, a fourth data set was chosen to feature the effects of knot insertion and removal.
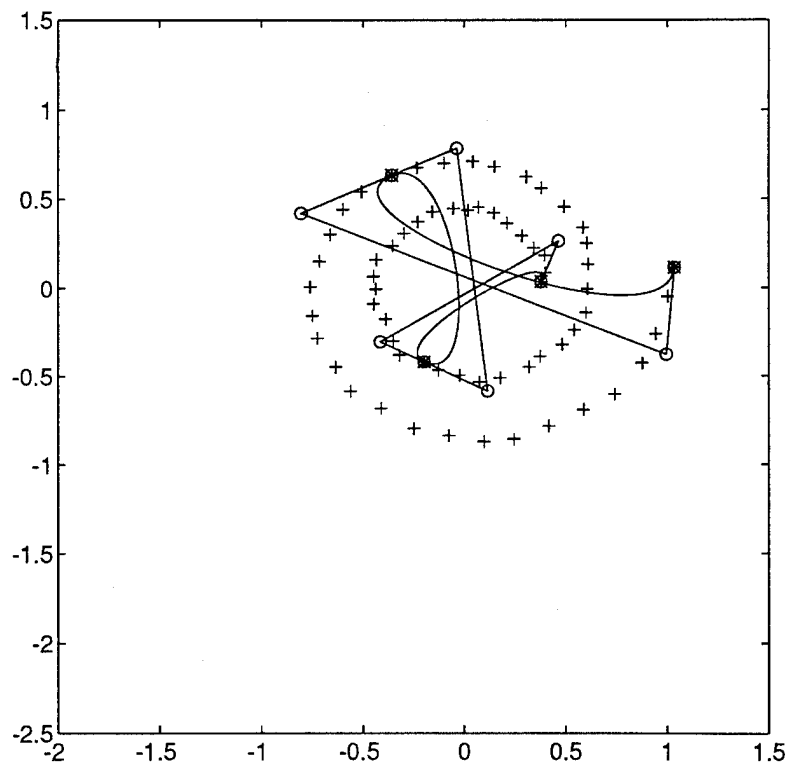

Figure 8. IG curve, k = [1 22 43 64].

The first data set contained 64 data points forming a spiral. It presents a problem similar to that of Marin and Smith (1994) in fitting a parametric curve to analytically represent the shape of a cross section of a machinery component. Figure 8 is the IG curve. The number of knots in the knot sequence is a result of trial and error in finding the minimum, in this case 4, which will later yield a "good" fit to the data set. It is clearly not a good fit by any means. However, as will be seen in a moment, it is a pretty good starting point. The rms error was 0.5042 units.
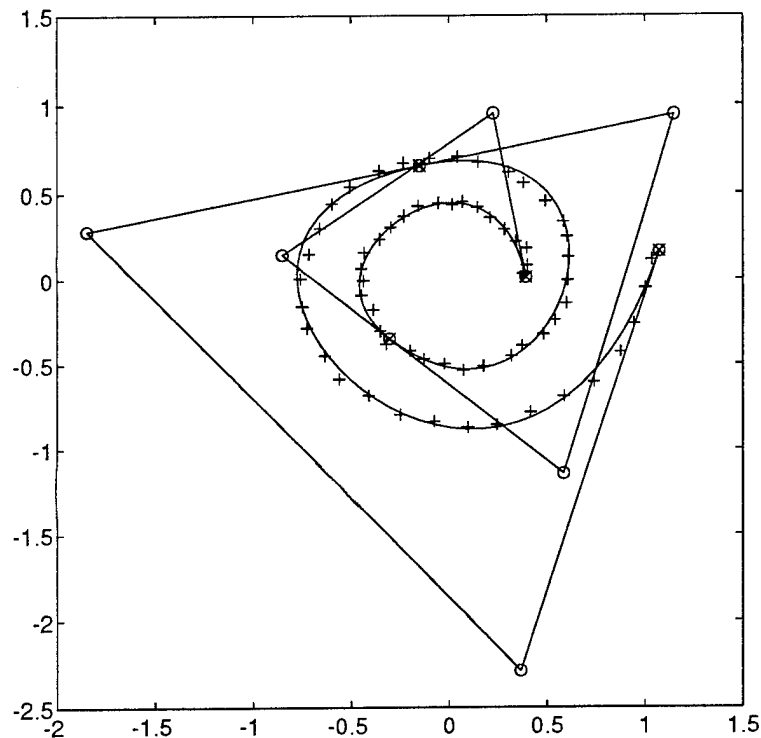
Figure 9. GOO curve, k = [1 21 41 64].

Figure 9 is the GOO curve. The fit is arguably reasonable and representative of the data set. The rms error is 0.0224 units. We observe that the curve meanders in and out of the path of the data points on the outer ring of the spiral while closely tracking the inner ring. Also noted, is the movement of the interior knot points.
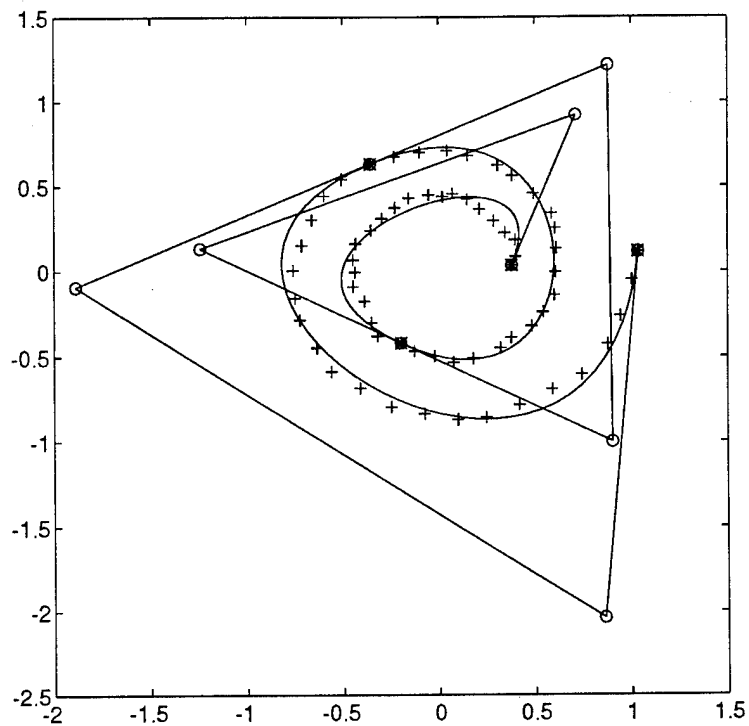
Figure 10. SOO curve, k = [1 22 43 64].
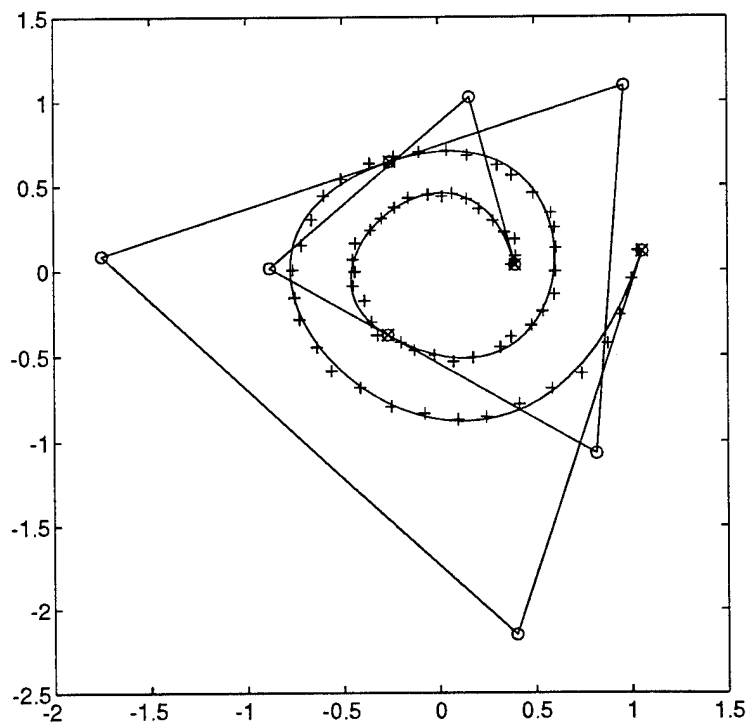
Figure 11. SGO curve, k = [1 21 42 64].

Figure 10 is the SOO curve. This is verified by checking the knot locations are the same as in the IG curve. The rms error is 0.0400 units and is representative of the "slack" observed between the data points and curve. Figure 11 is the SGO curve and its rms error is 0.0205 units. We see the curve tracks along the overall path of the data points more closely than the curve in Figure 9. Note: the algorithm converged to a solution at all stages.
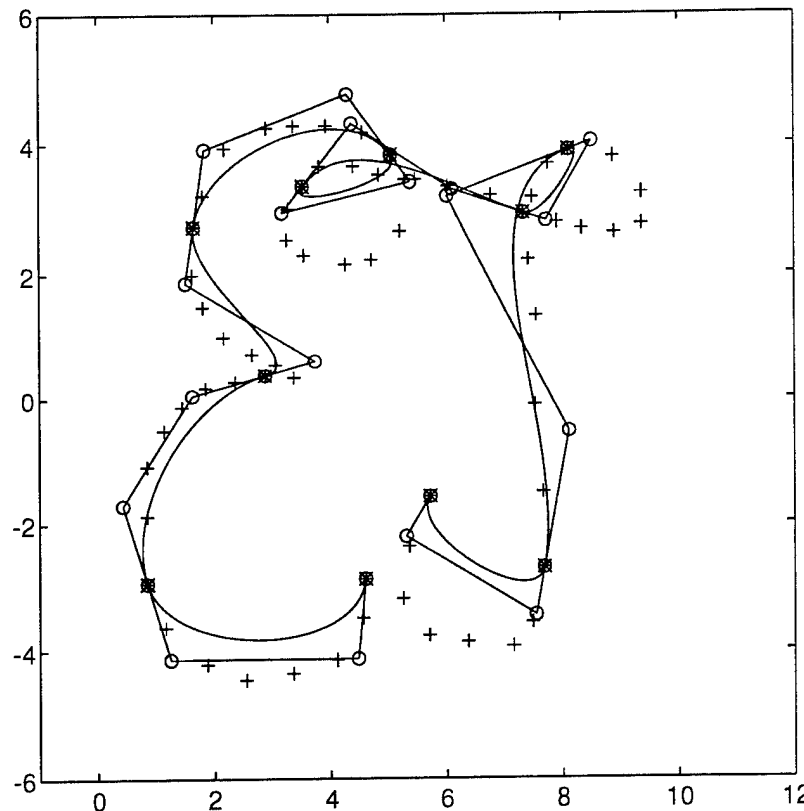
Figure 12. IG curve, k = [1 8 15 22 29 37 44 51 58 65].

The second data set is 65 points forming the letters "EJ". It presents the difficulties of multiple loops and a naturally formed cusp. The IG curve with 10 knots is displayed in Figure 12. The fit is fairly consistent with the trends in the data and has an rms error of 0.3125 units. We see the curve has no cusps, corners, or kinks.

Figure 13 is the GOO curve. The rms error is 0.0586

units but the curve is unsatisfactory. The problem area is an undesirable cusp in the top loop of the "J". This was caused by two control points having what appears to be coincident tangents in the same direction out of their common knot point.
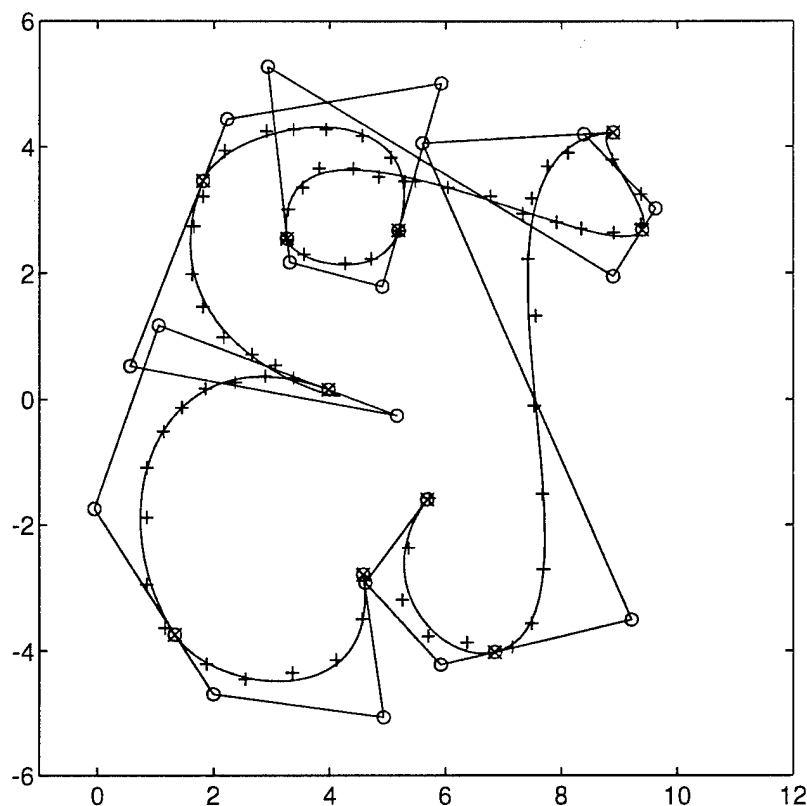
Figure 13. GOO curve, k = [1 6 16 18 31 35 43 50 59 65].

The SOO curve appears in Figure 14. For the most part, the curve tracks the data points nicely. It has an rms error of 0.0642 units. We see the desired cusp is forming in the middle region of the "E". Also, we see a problem area in the loop at the top of the "E". This is a kink or "cornering" effect due to near coincidence of a knot and control point and the pulling effect of the adjacent control point.
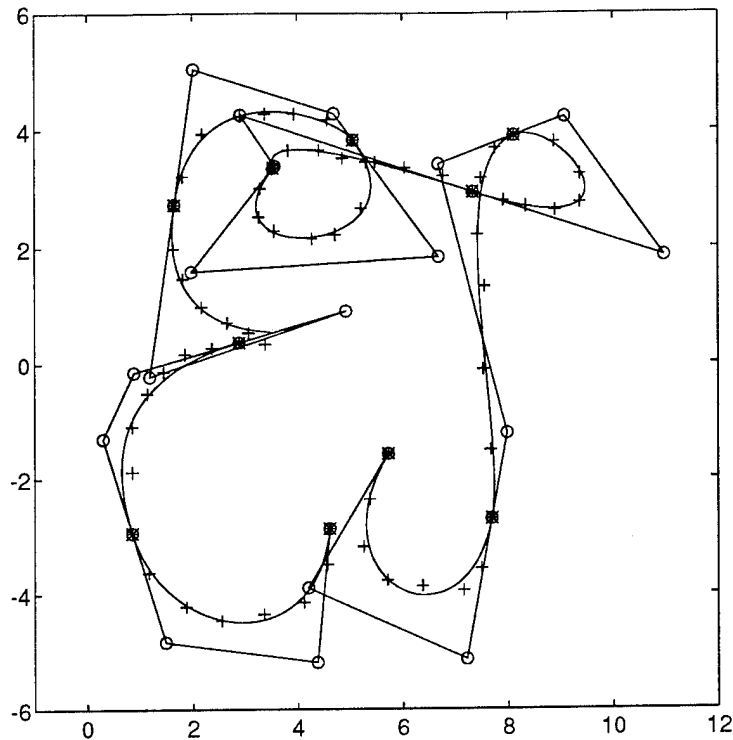
Figure 14. SOO curve, k = [1 8 15 22 29 37 44 51 58 65].
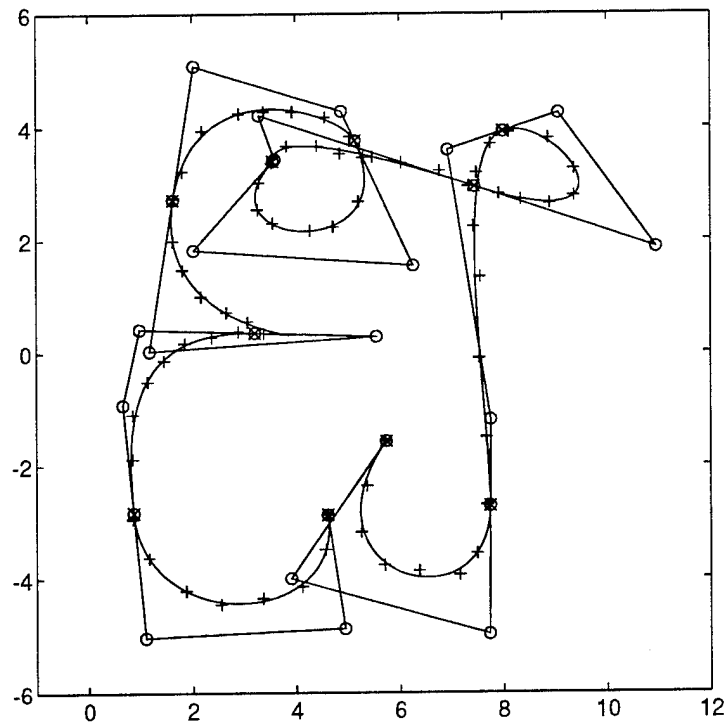
Figure 15. SGO curve, k = [1 8 16 22 29 37 44 50 58 65].

Figure 15 is the SGO curve fitted. The curve again tracks the data nicely. Its rms error is 0.0369 units. We see in the upper loop of the "E", where the problem area was in the SOO curve, the cusp or "cornering" still exists but is diminished somewhat by the movement of the adjacent control point. Note: the algorithms converged in all stages except for the GOO curve.
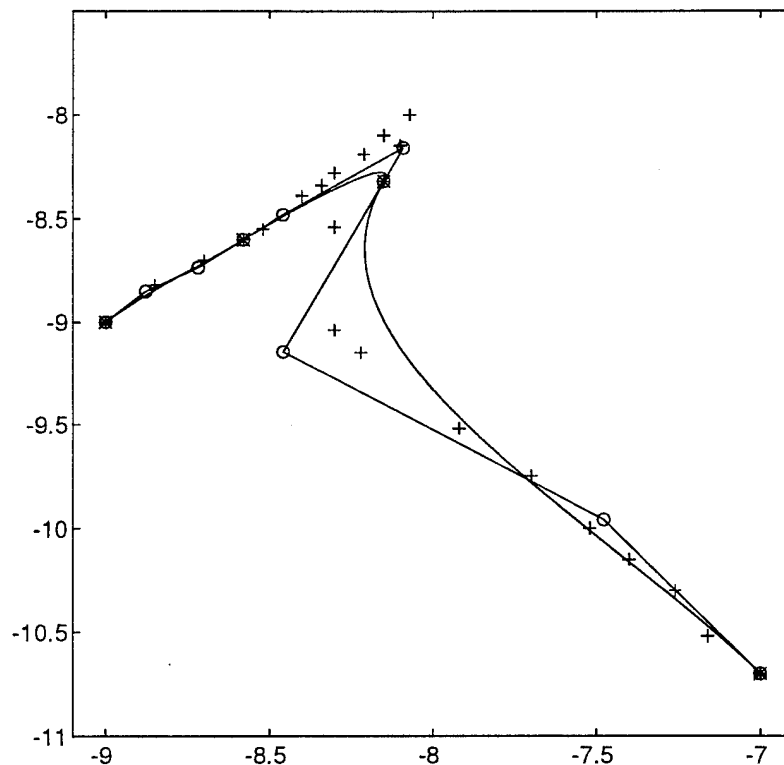
Figure 16. IG curve, k = [1 4 13 23].

The third data set contains 23 points and presents the unique demands of fitting some data found in a laboratory experiment on a reacting chemical system with potential multiple steady states. The experiment samples the steady state oxidation rate R achieved by a catalytic system for an input concentration of carbon monoxide C_{co}. The resulting data is plotted as log-vs-log. For more information see (Marin, 1994).

Figure 16 shows the IG curve found for the data set using 4 knots. The curve captures the trend of the data points and has an rms error of 0.0934 units. Figure 17 shows the GOO curve. The rms error is 0.0449 units with the second segment making the most contribution. We see that the peak of the curve appears to form a cusp and is short of the highest data point and that many data points are missed.
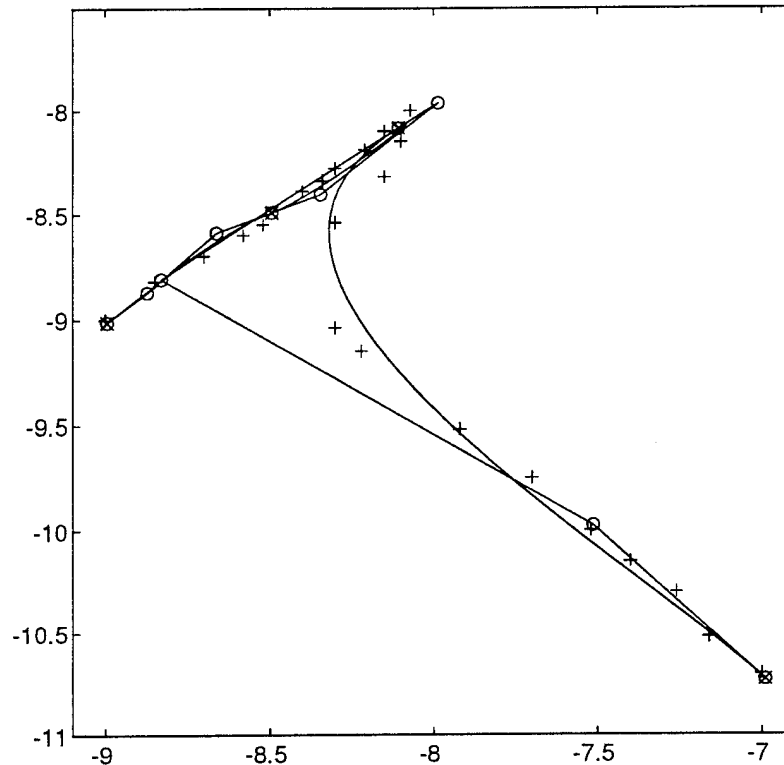
Figure 17. GOO curve, k = [1 5 10 23].

We next see the fit of the SOO curve in Figure 18. It has an rms error of 0.0177 units. We see the curve is following the path of the data nicely and misses very few. Finally, Figure 19 displays the SGO curve. The rms error is 0.0114 units and, as can be seen, the curve is a "good" fit to the data points.
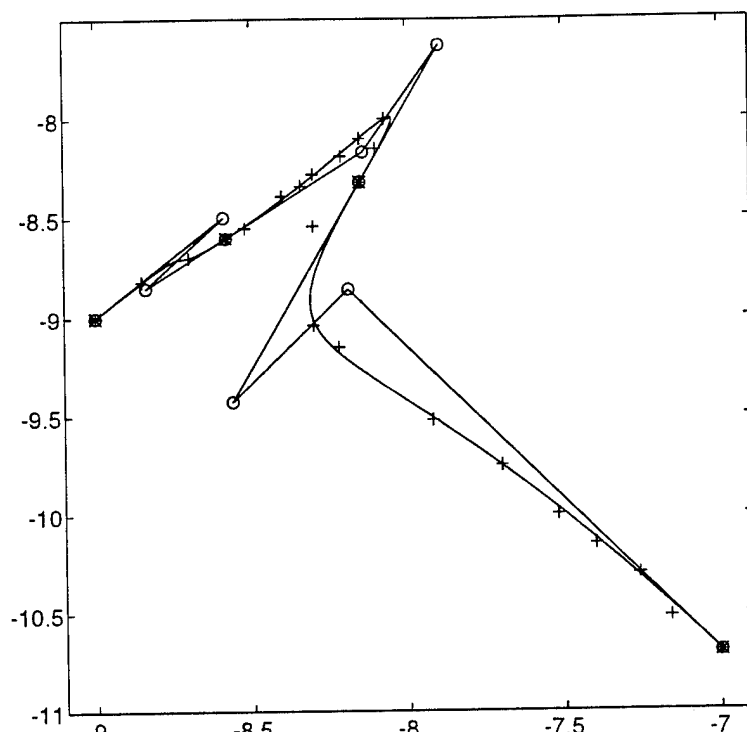
Figure 18. SOO curve, k = [1 4 13 23].
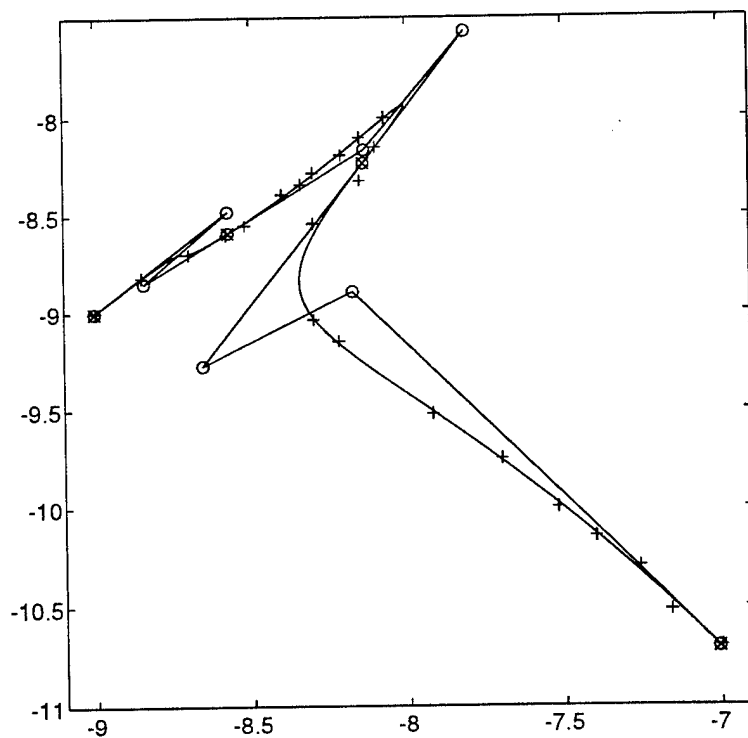
Figure 19. SGO curve, k = [1 4 12 23].

51

1. Knot Insertion and Removal

We now look at a data set of 23 points representing a single loop. Figure 20 is a SOO curve for the data set. It captures the shape of the data set rather well and has an rms error of 0.2492 units. Although this curve would likely lead to a "good" fit, we want to alter the knot sequence by inserting and deleting some knots.
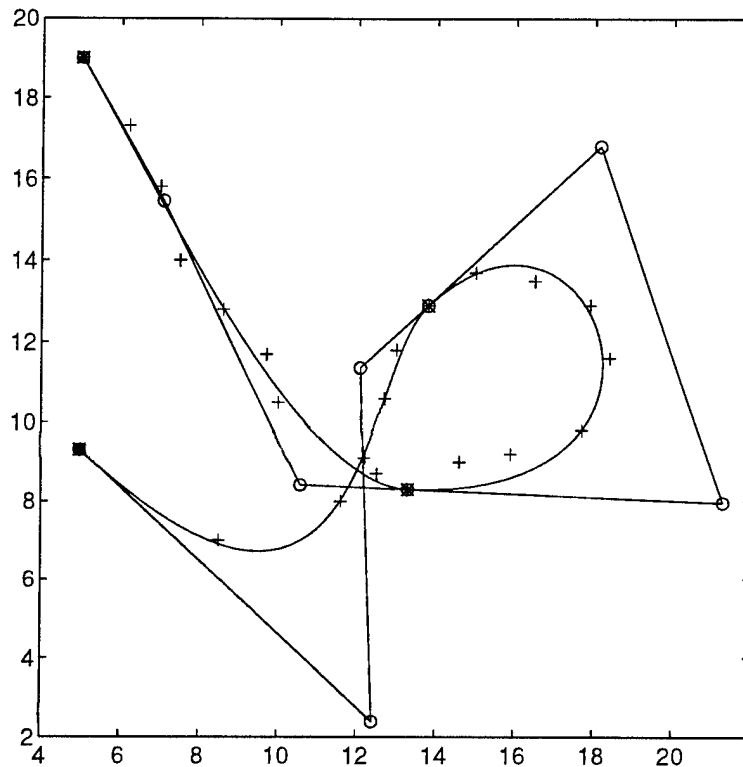
Figure 20. SOO curve, k = [1 9 17 23].

We believe the curve could potentially fit the second segment better. Therefore, we insert another two knots along the second segment. Further, we insert one knot on the third segment to facilitate removing the knot at position 17. We then remove the two original interior knots. Figure 21 is the resulting curve. We see the most change to the curve occurs along its lower path. This is because more control was placed along the top of the curve while it was relaxed at the bottom. The rms error is 0.6587 units.
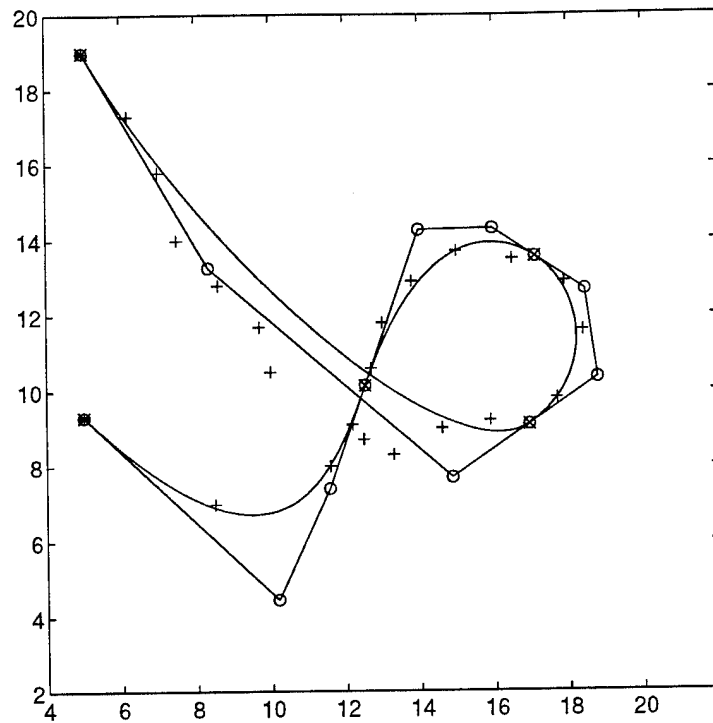
Figure 21. Altered curve, k = [1 12 15 19 23].
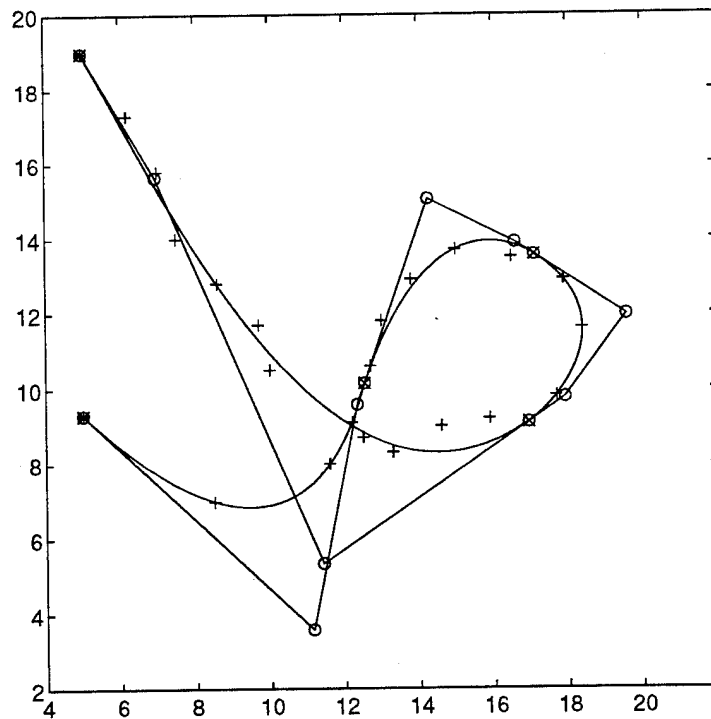
Figure 22. SOO curve, k = [1 12 15 19 23].

53

Figure 22 is the SOO curve. The fit is better and has an rms error of 0.2339 units. We now globally optimize this curve.
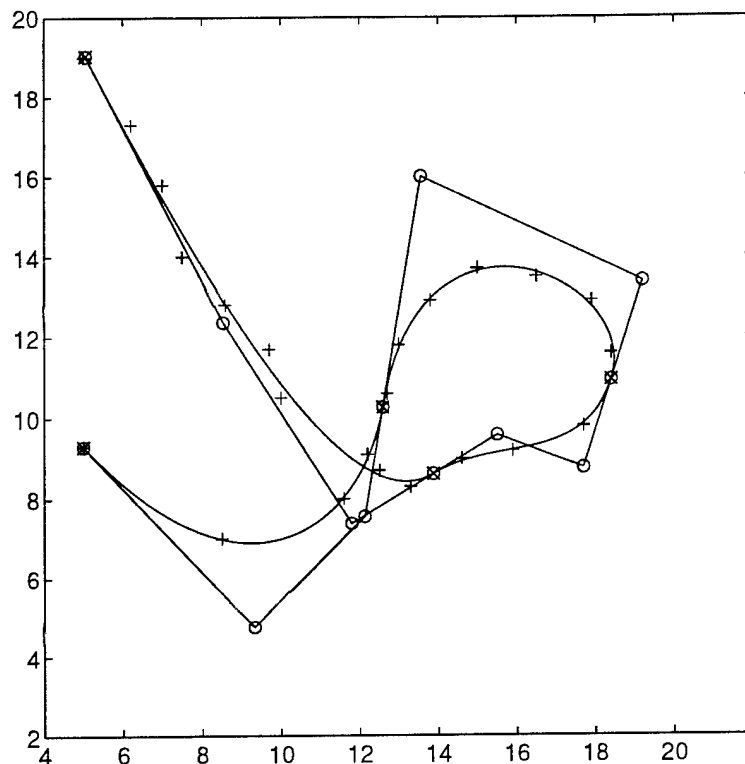
Figure 23. SGO curve, k = [1 9 13 19 23].

Figure 23 is the SGO curve. It is a "good" fit and has an rms error of 0.1249 units. We see the second and third knots moved quite a bit in the global optimization stage.

B. CONCLUSIONS AND RECOMMENDATIONS

The method shows promise of being able to fit a set of ordered data with a "good" approximating curve with minimal user interaction. Some recommendations toward reaching this goal are: an improved knot selection routine, improvements in the knot insertion and removal routines, implementing an affine invariant metric on the objective functions for optimization, and gearing the optimization routine more toward the problem at hand.

The current algorithm works well provided a good sequence of knot positions are selected in the initial guess stage. The number of knots and their positions is a function of the complexity of the underlying the relationship between the data points and the shape limitations of using cubic Bézier curves. For example, in a given cubic Bézier segment, the curve can have at most one loop, one point of inflection, one cusp, or one "corner". Hence, if a data set had a loop and cusp along its ordered path, a minimum of six knots would be required for an adequate fit. Therefore, a routine could be implemented that accounts for maxima, minima and variations in the data when selecting the knot points.

The knot insertion and removal routines are limited to one insertion or deletion at a time. These routines could be altered to allow multiple changes to occur simultaneously. Further, the removal routine could be improved so that it reproduces the original curve more closely.

The current objective functions for the optimization process rely on orthogonal distances. Since orthogonality is not affinely invariant, a metric could be induced like that of Nielson (1987) which would make the objective functions affinely invariant.

The optimization routine sometimes converges to an undesirable solution (i.e. the curve has kinks and cusps), or converges slowly, or does not converge at all. The problems of kinks and cusps could possibly be cleared up by use of some "penalty" terms in the objective functions when these conditions are encountered and are not desirable. The convergence problems could be improved by implementing an optimization routine based on nonlinear least squares fitting techniques.

[xi, k] = iguess[Q];

iguess.m

k

P

dt

ang

C

user man. input
knot posits.

defk.m

knots.m

dist.m

tang.m

uv

unitv.m

ctpts.m

pltC.m

berny.m

plot of curve

xi = [P ang dt];
and k .
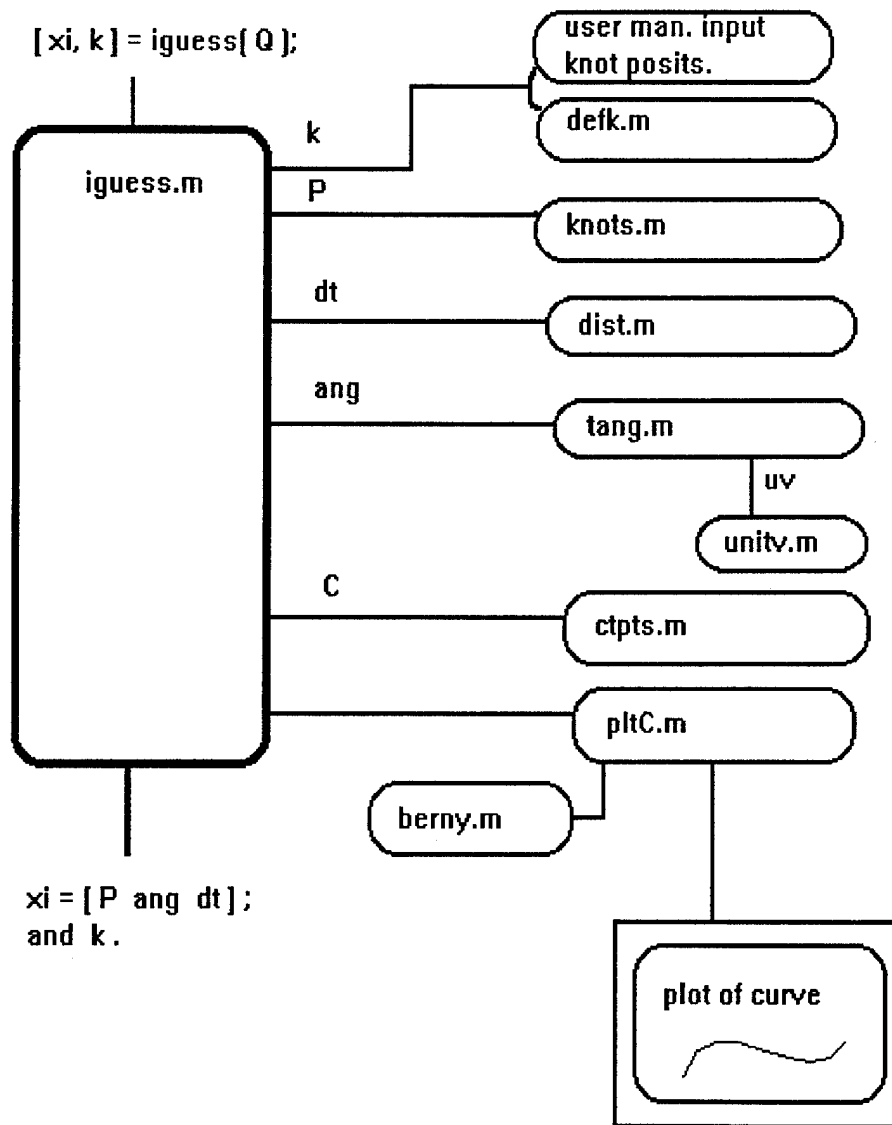
Flow Chart 1. Initial Guess

xi = segop[k ,Q , x0];

segop.m

P, ang, dt

ktangdt.m

bdt

bstdst.m

bdt (one seg. @ a time]

fmins

seg. error

opdist.m

np

NearestPoint

berny.m

ctpts.m

pltC.m

xi = [P ang dt];

C

poplt[xi, Q];

poplt.m

plot of curve

ktangdt.m

P, ang, dt
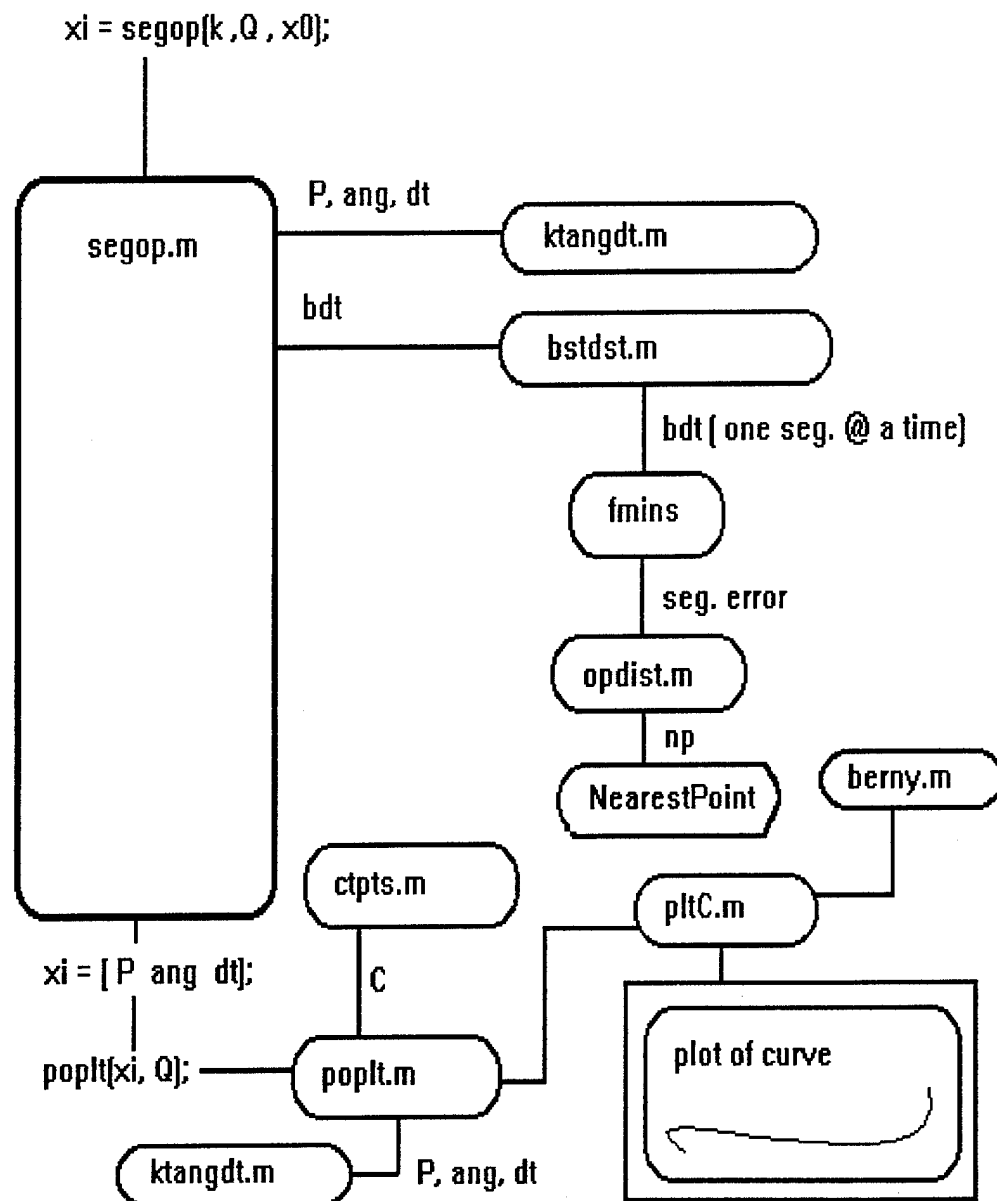
Flow Chart 2.   Segment Optimization

58

xi = globop [x0, Q, t, k0];

```
             ┌──────────┐
             │  fmins   │
             └──────────┘
                 │ error sum
┌──────────┐   ┌──────────┐
│          │   │ objf2.m  │──── P, ang, dt
│          │   └──────────┘
│ globop.m │        │      ┌──────────┐
│          │        │      │ ktangdt.m│
│          │      C │      └──────────┘
│          │        │   ┌──────────┐
│          │        │   │ ctpts.m  │
│          │   dpkpc│   └──────────┘
│          │        │ ┌──────────┐
│          │        └─│ newk.m   │
└──────────┘          └──────────┘
```

xi = [P ang dt];

sum error
for curve

```
                    ┌──────────┐
                    │  sod.m   │
                    └──────────┘
                         │ np
poplt [ xi, Q ];    ┌──────────────┐
                    │ NearestPoint │
┌──────────┐        └──────────────┘
│ poplt.m  │
└──────────┘
```

plot of curve
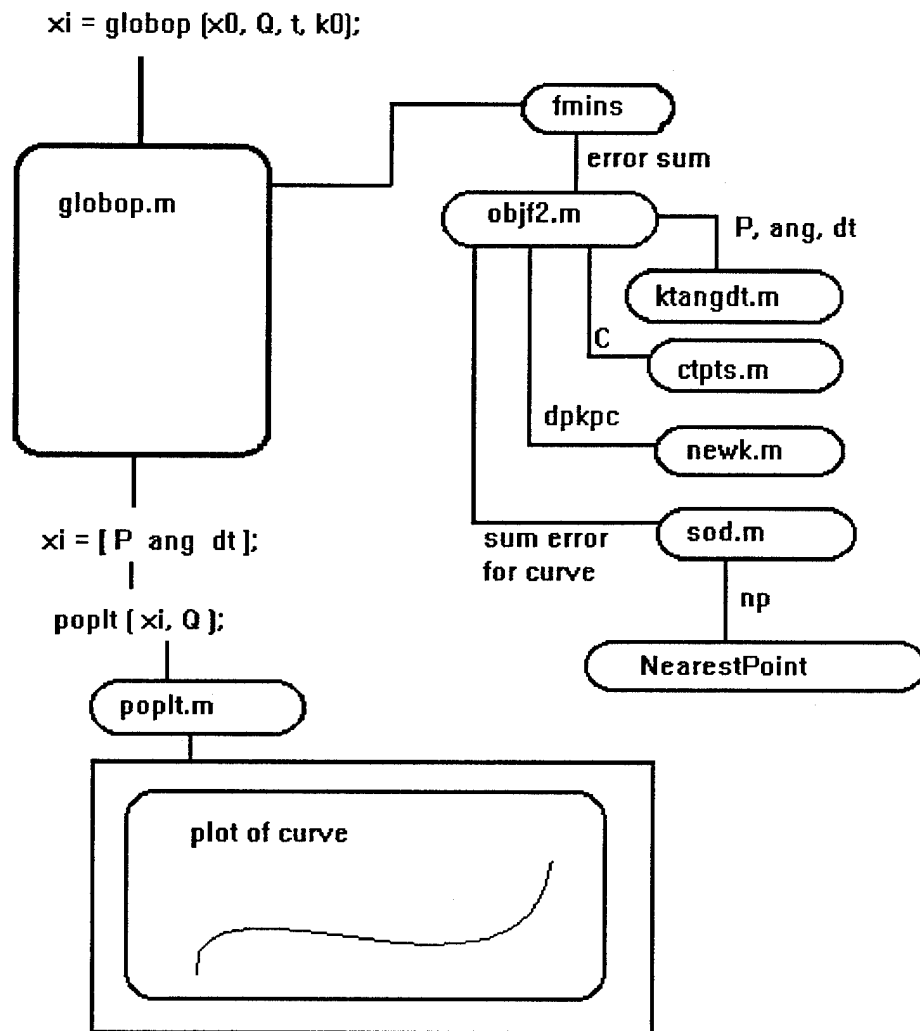
Flow Chart 3. Global Optimization

```
function [IG,k] = iguess(Q)

% function [IG,k]=iguess(Q). This routine takes a set of data points, Q;
% picks out a subset of the data points for knot points,P; computes the
% position of the knot points,k; computes the initial distances, dt, to
% place the interior control points, C, which are also computed; computes
% the angles, ang, of the unit tangent vectors at each knot point; and
% assembles the vector IG of parameters  P, ang, and dt, for the curve.
% The routine returns the "vector" of parameters and plots the curve,
% its polygon, and the data points in Q.  It was written  by M. R. Holmes
% and revised by E. J. Lane.

 global dpkpc;

 [r,m] = size(Q);

 disp('Give the number of knotpoints.')

   n = input('  ');

 disp('Type "1" for default knot position or "2" to input your own.')

     h = input('  ');

 if h == 1

     k = defk(m,n);   % Calls for default knot position.

     elseif h == 2

    disp('Input initial knot sequence as follows "[1 4 8 ...n]".')

          k = input('  ');

 elseif h ~= 1 | h ~= 2

     disp('Error! Start over and choose "1" or "2".'),pause(2)

         iguess
 end

 dpkpc = k; % Position of knot points passed globally.

   P = knots(Q,k); %  Call to compute the knotpoints.

  dt = dist(P);      % Call to compute the distance between
                     % successive knot points.

 ang = tang(Q,k);   % Call to compute the angles for
                    % the unit tangent vectors.

   C = ctpts(P,ang,dt);  % Call to compute the control points
                         %for the curve.

  pltC(C,Q,P); % Call to plot the initial guess curve,
               % its control polygon, and points in Q.

 IG = [P(1,:) P(2,:) ang dt(1,:) dt(2,:)];
 % Assemble the composite vector of the initial guess curve parameters.
```

```
function k = defk(m,n)

% Computes default knot positions for iguess.m based on a
% formula to equally disperse the knots throughout the data.

k = round((((m-1)/(n-1))*[0:n-1] + ones(1,n));
```

```
function P = knots(Q,k)

% function P = knots(Q,k). This function takes data points Q
% and knot sequence vector k and picks out the knot points
% of the curve.

   P=[];   P=[P Q(:,k)];
```

```
function dt = dist(P)

% function dt = dist(P).  This function computes the initial
% distances from the knot points to their adjacent control
% points for the initial guess curve. It returns the vector
% of distances to iguess.m.  The function was written by
% E.  J.  Lane.

   t=length(P);

   d1 =  P(:,1:t-1) - P(:,2:t); % Calculates inter-knot
                                % x and y difference values..

   d2 = sqrt(sum(d1.^2))/3;  % Computes the initial distances.

   dt = [d2;d2]; % Assembles the vector of distances.
```

61

```
function ang = tang(Q,k)

% function ang = tang(Q,k).  This function computes the angles
% for the unit tangent vectors at the knot points.

u = unitv(Q,k); % Call to compute unit tangents.

ang = atan2(u(2,:),u(1,:)); % Converts tangents to angles.
```

```
function uv = unitv(Q,k)

% function uv = unitv(Q,k).  This function takes data points,
% Q, and the position of knotpoints, k, as input variables.
% It uses chord length parameterization to fit a parametric
% quadratic curve to five data points.  The unit tangent vec-
% tors are approximated by the unit tangent vectors for these
% quadratic functions.  It returns the set of unit tangent
% vectors in the direction of the knot points.  It was written
% by M. R. Holmes

  [r,m] = size(Q);   n = length(k);

  for j = 1:n                    % Loop to index knot positions.
      if j == 1, k(j) = 1; kt = 1;
      elseif j == n, k(j) = m-4; kt = 5;
      else k(j) = k(j)-2; kt = 3;
      end

  x = Q(1,k(j):k(j)+4)';    % Extracting the knot point
  y = Q(2,k(j):k(j)+4)';    % and four adjacent points.

  xd = diff(x);  yd = diff(y);  % Get chord length.
  d = sqrt( xd.*xd + yd.*yd);
  t(1) = 0; t(2) = d(1);
  t(3) = t(2) + d(2);
  t(4) = t(3) + d(3);
  t(5) = t(4) + d(4);

  c = [ones(5,1)  t'  (t.*t)'] \ [x  y];
  u = c(2,:) + 2*c(3,:)*t(kt);  u = u/norm(u);
  uv(:,j) = u';                  % Approximation of unit tangents
                                 % by  unit tangent to quadratic.
  end
```

62

```
function C = ctpts(P,ang,dt)

% C = ctpts(P,ang,dt).  This function takes knot points,P; angles
% of the tangent vectors, ang; distances between successive knot
% points, dt; as input.  It then computes the positions for the
% control points.  It was written by M. R. Holmes.

 n = length(P);

  for k = 2:n-1

  u = [cos(ang(k))  ; sin(ang(k))];
  % Converts the interior angles into their x and y components.

  T = [T  P(:,k)-u*dt(2,k-1)  P(:,k)   P(:,k)+u*dt(1,k)];
        % Assembles the vector knot points with their
        % adjacent interior control points.

  end

  u1 = [cos(ang(1))  ; sin(ang(1))];  % Converts the first and last
  un = [cos(ang(n))  ; sin(ang(n))];  % angles into their x and y
                                      % components.

  C = [P(:,1)   P(:,1)+u1*dt(1,1)   T   P(:,n)-un*dt(2,n-1)   P(:,n)];
        % Assembles the vector of all control points.
```

```
function graf = pltC(C,Q,P)

% function graf = pltC(C,Q,P).  This function takes as input:
% control points,C, data points,Q; and knot points, P. The
% control points are used to calculate the points of the
% approximating cubic Bézier curves.  The control, data, and
% knot points are then plotted along with the curve.
% This was written by M. R. Holmes.

  [s,t] = size(C);
   x = [0:.025:1]; % Defines the interval for the polynomial.
  [a,b] = size(x);

  W = [ ];                % Loop to construct the Bézier curve.
for j = 1:3:t-3
   Y = zeros(2,b);
  M = [berny(3,0,x)' berny(3,1,x)' berny(3,2,x)' berny(3,3,x)'];
   Y = Y + C(:,j:j+3) * M' ;
   W = [W  Y];
   end


  plot( W(1,:) , W(2,:) ) , hold

  plot( C(1,:) , C(2,:) )

  plot( Q(1,:) , Q(2,:) , '+')

  plot( P(1,:) , P(2,:) , 'x')

  plot( C(1,:) , C(2,:) , 'o')
```

```
function val = berny(n, i, x)

% This function is a non-recursive formula for cubic Bernstein
% Polynomials which form the basis for the cubic Bézier curves
% which are used in the supporting programs.  The inputs are the
% degree of the polynomial, n; the particular curve that is ass-
% igned a value of zero up to and including the degree, i; and
% the points between [0,1] to be evaluated, x.  The output is
% points on the curve.  It was written by M. R. Holmes.

   ni = [1 3 3 1];   m = size(x);

   if n < i
      val = zeros(m);
   elseif i < 0
      val = zeros(m);
   elseif ((n == 0) & (i == 0))
      val = 1;
   else
      val = ni(i+1) * (x.^i) .* ((ones(m) - x) .^(n-i));
   end
```

```
function SOC = segop(k,Q,x0)

% function SOC = segop(k,Q,x0).  This function returns the
% parameters for the segmentally optimal composite curve.
% It receives the .IG curve parameters, x0, data points Q,
% and knot sequence k. It was written by E. J. Lane.

 [P,ang,dt]=ktangdt(x0); % Separates the vector x0
                         % into its subcomponents.

 bdt=bstdst(dt,Q,P,ang,k); % Call to the function which finds
                           % the optimum distances for a segment.

  for i = 1 : 2    % Loop to assemble the "best" distances.

     bdt1 = [bdt1 bdt(i,:)];

  end

SOC = [P(1,:) P(2,:) ang bdt1];
                    % Assemble the vector of parameters
                    % for the curve.
  end
```

```
function [P,ang,dt] = ktangdt(x)

% function [P,ang,dt] = ktangdt(x).  This handy function separates
% the composite vector, x, of parameters for a curve into the sub-
% components of knots, P, angles, ang, and distances, dt. It was
% written by E. J. Lane.

  m = length(x);   n = round(m/5);

  P(1,:) = x(1:n);        % knots.
  P(2,:) = x(n+1:2*n);

  ang = x(2*n+1:3*n);   % angles.

  dt(1,:) = x(3*n+1:4*n-1);
  dt(2,:) = x(4*n:m);          % distances.
```

```
function bdt = bstdst(id,Q,P,ang,k)

% This function finds the optimum distances for control point
% placement along the segments of a curve. The applicable points
% from Q, the two knots, and two angles for each segment are
% passed to opdist.m through "fmins". It was written by E. J. Lane.

  opts = [0,.01,.01]; % Control parameters for "fmins".

  n = length(id);   bdt=[];

  for i = 1 : n

bdt(:,i)= fmins('opdist',id(:,i),opts,[],...
                ...Q(:,k(i):k(i+1)),P(:,i:i+1),ang(i:i+1));

  end
```

```
function se2 = opdist(id,Q,P,ang)

% function se2 = opdist(id,Q,P,ang).  This function is the obj-
% ective function to be minimized during segment optimization.
% It receives subcomponents from a curve's composite vector, a
% segment at a time.  It returns the sum error for a segment.
% It was written by E. J. Lane.

 n = length(Q);

 C=ctpts(P,ang,id); % Call to compute the segments
                    % control points.
 se2=0;

 for j = 1 : n                          % Loop to find
        np = NearestPoint( C' , Q(:,j)');  % distance error
                                         % in a segment.
        if j==1 &  np==C(:,1)'
           d=zeros(1,2);
        elseif j==n & np==C(:,4)'
           d=zeros(1,2);
        else
           d = (Q(:,j)' - np);
        end
        se2 = se2 + d*d';
 end
```

```
function pop = poplt(x,Q)

% function pop = poplt(x,Q).  This function picks out subcomponents
% of the vector x of curve parameters.  It calls the function that
% computes the control points. It then calls for a plot of the curve
% its polygon, and the data points.  This was written by M. R. Holmes
% and revised by E. J. Lane.

 [P,ang,dt] = ktangdt(x); % Separate vector x.

 C = ctpts(P,ang,dt); % Call to compute the control points.

 pltC(C,Q,P) % Call to plot the curve, polygon, and data points.
```

67

```
function GOC = globop(xi,Q,t,k)

% function GOC = globop(xi,Q,t,k).  This function returns a vector
% GOC of parameters: knot points, P, angles, ang, and distances,
% dt, for a globally optimized Bézier curve.  Its inputs are
% the curve parameters in vector xi , data points, Q, toggle, t,
% "1" if a knot was inserted or removed, "0" otherwise , and the
% knot sequence, k. The MATLAB routine "fmins" optimizes function
% objf2.m which computes the sum of the distances between the data
% points and their closest point on the curve.  It was written by
% E. J. Lane.

 GOC = fmins('objf2',xi,[0,.01,.01],[],Q,t,k);
```

```
function se = objf2(x,Q,t,k)

% function se = objf2(x,Q,t,k).  This is the objective function that
% will be minimized by "fmins".  The input arguments are the vector x of
% parameters for the curve, data points Q, a toggle t if a new knot has
% been inserted or one removed, and the knot sequence, k. The output is
% the sum from the function "sod" plus the distance squared from the
% first and last data points to the first and last knot points, respec-
% tively.  This was written by M. R. Holmes and revised by E. J. Lane.

  global dpkpc

  if t == 1              % Loop to change dpkpc if a knot
     dpkpc = k; t = 0;   % was inserted or removed.
     global dpkpc
  end

  if t == 0

  global dpkpc

  [r,s] = size(Q);

 [P,ang,dt] = ktangdt(x); % Call to separate x into its subcomponents.

  C = ctpts(P,ang,dt); % Call to compute control points.

  dpkpc = newk(Q,P);    % Calls function that computes the
                        % new dividing point positions.

  m = length(x);
  n = round(m/5);

  fp = P(:,1) - Q(:,1); % Computes distance squared
  lp = P(:,n) - Q(:,s); % from the first and last data points to
                        % the first and last knot points, respectively.

  se = sod(C,Q,dpkpc) + fp'*fp + lp'*lp ;
            % Calls the function that computes the sums of the square
            % of the distances from the data points to the nearest point
            % on the cubic segment.

end
end
```

```
function  nk = newk(Q,P)

% function nk = newk(Q,P). This function takes data points, Q,and
% knot points, P, as input. The function finds the closest data
% point of the cubic segment that is associated with that knot
% point, and returns a new k-array, nk.  It ensures the data points
% of Q are properly associated with the proper segment of the curve.
% "dpkpc" is a global variable that is initially equal to the old
% k.  This was written by M. R. Holmes and revised by E. J. Lane.

  global dpkpc

  [r,m] = size(Q);

  [s,n] = size(P);

  nk(1) = 1;  nk(n) = m; % Ensures the knot sequence starts and
                         % ends with the 1st and last points in Q.

  for i = 2:n-1
      js = dpkpc(i-1); je = dpkpc(i+1); % variables to pick out
      jm = dpkpc(i);                     % interior knot positions.
      z =  je-js+1;  mm = jm - js + 1;
      R = Q(:,js:je) - P(:,i) * ones(1,z); % Finds differences
                                  % between data points and
                                  % knot point being checked.

      for jj = 1:z
          D(jj) = R(:,jj)' * R(:,jj);  % Ensures differences are
      end                              % positive for comparison.

      if mm < z
          sd = sign( D(mm) - D(mm+1) ); % Compares for smallest
                                        % difference to find
        elseif mm > 1                   % new dividing points.
              sd = sign(D(mm-1) - D(mm));
        else
              sd = 0;
      end

      while  D(mm) - D(mm+sd) > 0
         if mm == 2 & sd < 0
             break, end

        ,if mm == m-1 & sd > 0
             break, end

         mm = mm + sd;
      end

      nk(i) = mm + js - 1; % knot positions.
  end

  dpkpc = nk; % knot positions or sequence.
```

```
function sumd = sod(C,Q,dpkpc)

% function sumofdist = sod(C,Q,dpkpc). This function receives inputs:
% control points,C, data points,Q, and the dividing points or knot
% sequence.  It finds the closest point on the curve for a given data
% point  and computes the distance error.  The function returns the
% sum of the distance squared from the data points to their nearest
% point on the curve segment.  This was written by M. R. Holmes.

   n = length(C);   [r,s] = size(Q);

   y = dpkpc;
   cntr = 0;
   sum = 0;
                                   % Loop to find distances from
   for i = 1:3:n-3                 % data points to closest point
       cntr = cntr + 1;           % on the curve.
       for j = y(cntr):y(cntr+1)
          np = NearestPoint( C(:,i:i+3)', Q(:,j)');
          d = ( Q(:,j)' - np );
          sum = sum + d * d';      % Note: NearestPoint is a MATLAB
          if j == y(cntr) &  i>1   % interface program written by
             d2 = d*d';            % Dr C. Borges for some "C" rou-
             ds2 = ds*ds';         % tines obtained from 'Solving
             dm = max(d2,ds2);     % the Nearest Point-on-Curve
             sum = sum - dm;       % Problem' and 'A Bézier Curve
          end                      % Root-Finder' developed by P.J.
      end                          % Schneider in "Graphics Gems",
      ds = d;                      % Academic Press, 1990.
   end

   sumd = sum ;
```

```
function error =err(x0,Q,k)

% function error = err(x0,Q,k).  This function takes a composite
% vector of curve parameters x0,  separates them and computes the
% control points for the curve.  It then computes the sum of the
% error between the curve and the data points in Q.   It was
% written by E. J. Lane.

   [P,ang,dt]=ktangdt(x0); % Call to separates x0
                           % into its subcomponents.

   C=ctpts(P,ang,dt); % Call to compute control points.

   error=sod(C,Q,k); % Call to compute distance
                     %error for the curve.
```

```
function [xi,nk] = insrtkt(seg,h,x0,k,Q)

% function [xi,nk] = insrtkt(seg,h,x0,k).  This function receives the
% segment number, seg, to have the knot inserted, the position along
% the segment, h, where it will be inserted, and the vector, x0, of
% parameters for the curve, and k, the knot positions.  It inserts a
% new knot on the segment called for and then returns the new vector
% of parameters for the curve and knot positions.  Note: the curve
% will remain the same, the polygon will be changed.  This was
% written by E. J. Lane.

 [P,ang,dt] = ktangdt(x0); % Separates x0 into its subcomponents.

 q = length(k);

 Cseg = ctpts(P(:,seg:seg+1),ang(seg:seg+1),dt(:,seg)); % Computes
             % the control points for the affected segment.

 z=fndpts(Cseg,h); % Call to compute new control points for the
             % segment where the knot is inserted.

 xs=z(1,:); ys=z(2,:); % Separates the new segment's control
                   % points into their x and y components.

 dx=diff(xs); dy=diff(ys); % Finds the intercomponent differences.

 angs=atan2(dy,dx); % Computes the angles for the tangent vectors.

 d1=sqrt(dx(1)^2 + dy(1)^2);          % Computes distances for
                                      % control point locations.
 de=sqrt(dx(6)^2 + dy(6)^2);

 dm=sqrt(dx(4)^2 + dy(4)^2);

 dn=sqrt(dx(3)^2 + dy(3)^2);

 Pnew = [P(:,1:seg) z(:,4) P(:,seg+1:length(P))];
                   % Inserts new knot into knot component vector.

 angnew = [ang(1:seg) angs(4) ang(seg+1:length(ang))];
             % Inserts new angles into tangent angles component vector.

 dtnew = [dt(:,1:seg-1) [d1 dm;dn de] dt(:,seg+1:length(dt))];
             % Inserts new distances into distance component vector.

 dv = Q(:,k(seg):k(seg+1)) - z(:,4)*ones(1,k(seg+1)-k(seg) + 1);

 ds =dv.*dv; dq=sum(ds); [dmin,knew]=min(dq);

 ink = k(seg) + knew - 1;        % With previous 2 lines
                                 % finds the new knot's position.
 nk = [k(1:seg) ink k(seg+1:q)]; % New knot sequence.

 xi = [Pnew(1,:) Pnew(2,:)  angnew dtnew(1,:) dtnew(2,:)];
                   % Assembles the new components vector for the
                   % parameters for the curve.
```

```
function x = fndpts(z,h)

% function x = fndpts(z,h).  The inputs are a vector z of control
% points for a segment of a curve and a step size h.  The function
% separates the control points into their x and y components and
% then uses a de Casteljau or Chaikin scheme to compute new control
% points which will produce the same curve.  It was written by
% E. J. Lane.

[m n]=size(z);  M=zeros(n);  N=zeros(n);

M(:,1) = z(1,:)';  % Separates the control points
N(:,1) = z(2,:)';  % x and y values.

for j = 2 : n           % Loop which performs the computation
                        % of new control point x and y values.
      for i = j : n

          M(i,j) = M(i-1,j-1) + ((M(i,j-1) - M(i-1,j-1))*h);
          N(i,j) = N(i-1,j-1) + ((N(i,j-1) - N(i-1,j-1))*h);

   end
end

x=[diag(M)' (rot90(M(n,1:n-1)))'; diag(N)' (rot90(N(n,1:n-1)))'];
  % Assembles the vector of new control points.
```

:

```
function [xi,nk] = rmvkt(kt,x,k)

% function [xi,nk] = rmvkt(kt,x,k). This function takes inputs of
% which knot, kt, to remove, vector, x, of curve parameters, and kt
% sequence, k.  It removes the knot, its angles, and its distances
% from the subcomponents of x, removes the index of the removed
% knot from k, then finds the knots that were adjacent to the one
% being removed, and constructs a new polygon for the "blended"
% curve segment.  This was written by E. J. Lane.

  [P,ang,dt] = ktangdt(x); % Separates the components of x.

  n = length(P);

  Pnew=[P(:,1:kt-1) P(:,kt+1:n)]; % Removes the knot.

  m = length(ang);

  angnew=[ang(:,1:kt-1) ang(:,kt+1:m)]; % Removes the knot's angles.

  p = length(dt);
  q = length(k);

  nk = [k(1:kt-1) k(kt+1:q)];  % Get rid of removed knot in sequence.

  Cseg=ctpts(P(:,kt-1:kt+1),ang(kt-1:kt+1),dt(:,kt-1:kt));
    % Computes the control points for the blended segment.

  xs=Cseg(1,:); ys=Cseg(2,:); % Separates the x and y components.
  dx=diff(xs); dy=diff(ys);   % Gets the differences in the x's, y's.
  dm=sqrt(dx(3)^2 + dy(3)^2);
  dn=sqrt(dx(4)^2 + dy(4)^2);   % Computes distances for the control
  dmn=dm+dn;                    % points on the blended segment.
  dt(1,kt-1)=dt(1,kt-1)*(dmn/dm);
  dt(2,kt)=dt(2,kt)*(dmn/dn);
                               % Assembles the distances.
  d1= dt(1,:); d2= dt(2,:);
  d11=[d1(1:kt-1) d1(kt+1:p)];   d22=[d2(1:kt-2) d2(kt:p)];
  dtnew=[d11 ; d22];

  xi = [Pnew(1,:) Pnew(2,:)  angnew dtnew(1,:) dtnew(2,:)];
    % Assembles the composite vector of parameters for the curve.
```

74

LIST OF REFERENCES

Burden, R. L., Faires, J. D., and Reynolds, A. C., <u>Numerical Analysis</u>, 2d ed, pp. 73-107, Prindle, Weber, & Schmidt, 1981.

Carnahan, B., Luther, H. A., and Wilkes, J. O., <u>Applied Numerical Methods</u>, pp. 1-40, John Wiley & Sons, 1969.

Cavaretta, A. S., and Micchelli, C. A., "The Design of Curves and Surfaces by Subdivision Algorithms", <u>Mathematical Methods in Computer Aided Geometric Design</u>, T. Lyche and L. Schumaker (eds), pp. 115-153, Academic Press, 1989.

Davis, P. J., <u>Interpolation and Approximation</u>, Blaisdell, 1963.

Farin, G., <u>Curves and Surfaces for Computer Aided Geometric Design</u>, 2d ed, Academic Press, 1990.

Foley, T. A., and Nielson, G. M., "Knot Selection for Parametric Spline Interpolation", <u>Mathematical Methods in Computer Aided Geometric Design</u>, T. Lyche and L. Schumaker (eds), pp. 261-271, Academic Press, 1989.

Gerald, C. F., and Wheatley, P. O., <u>Applied Numerical Analysis</u>, 4th ed, pp. 180-228, Addison-Wesley, 1989.

Hill, R. O. Jr., <u>Elementary Linear Algebra with Applications</u>, 2d ed, pp. 131-158, Harcourt Brace Jovanovich, 1991.

Holmes, M. R., <u>Least Squares Approximation by G^1 Piecewise Parametric Cubics</u>, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1993.

Lorentz, G., <u>Bernstein Polynomials</u>, 2d ed, pp. 3-51, Chelsea, 1986.

Marin, S. P., and Smith, P. W., "Parametric Approximation of Data Using ODR Splines", <u>Computer Aided Geometric Design</u>, vol. 11, pp. 247-267, 1994.

Mendenhall, W., Wackerly, D., and Scheaffer, R., <u>Mathematical Statistics with Applications</u>, 4th ed, PWS-KENT, 1990.

Nelder, J. A., and Mead, R., "A Simplex Method for Function Minimization", <u>Computer Journal</u>, vol. 7, pp. 308-313, 1965.

Nielson, G. M., "Coordinate Free Scattered Data Interpolation", <u>Topics in Multivariate Approximation</u>, C.Chui, L. L. Schumaker, and F. Utreras, (eds), pp. 175-184, Academic Press, 1987.

Pratt, M. J., "Parametric Curves and Surfaces as used in Computer Aided Design", <u>The Mathematics of Surfaces</u>, J. A. Gregory (ed), pp. 19-45, Clarendon Press, 1986.

Ralston, A., <u>A First Course in Numerical Analysis</u>, pp. 23-67 and 228-250, McGraw-Hill, 1965.

Rivlin, T. J., <u>An Introduction to the Approximation of Functions</u>, Dover ed, Dover, 1981.

Ross, K. A., <u>Elementary Analysis: The Theory of Calculus</u>, Springer-Verlag, 1980.

Schneider, P. J., "Solving the Nearest-Point-on-Curve problem", <u>Graphics Gems</u>, A. S. Glassner (ed), Academic Press, 1990.

Spendley, W., Hext, G. R., and Himsworth, F. R., "Sequential Application of Simplex Designs in Optimisation and Evolutionary Operation", <u>Technometrics</u>, vol. 4, p441, 1962.

The Math Works Inc., <u>MATLAB</u>, 1992.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, California 93943-5101	2
3.	Professor Richard Franke, Code MA/Fe Department of Mathematics Naval Postgraduate School Monterey, California 93943-5216	4
4.	Professor Carlos F. Borges, Code MA/Bc Department of Mathematics Naval Postgraduate School Monterey, California 93943-5216	1
5.	LT Edward J. Lane USS DWIGHT D. EISENHOWER CVN-69 FPO AE 09532-2830	2
6.	CAPT Christine E. Lane PSC 90, Box 1585 APO AE 09822	1
7.	Dr. Samuel P. Marin Mathematics Department General Motors Research Laboratories Warren, MI 48090-9055	1
8.	Professor G.M. Neilson Department of Computer Science Arizona State University Tempe, AZ 85287	1
9.	Mr. and Mrs. B.J. Lane 315 Southwalk Pl. Pensacola, FL 32506	1